WL-TR-94-8014

INTEGRATION TOOLKIT AND METHODS (ITKM), AUTOMATED PROTOCOL ANALYSIS/REFERENCE TOOL RESEARCH AND DEVELOPMENT PROJECT (APART)

FRANK J. WROBLEWSKI
J. DOUGLASS H. WHITEHEAD

CORPORATION FOR OPEN SYSTEMS INTERNATIONAL
8260 WILLOW OAKS CORPORATE DRIVE
SUITE 700
FAIRFAX VA 22031

MARCH 1994

FINAL REPORT

MANUFACTURING TECHNOLOGY DIRECTORATE
WRIGHT LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT PATTERSON AFB OH 45433-7739

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely related Government procurement operation, the United States Government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report has been reviewed by the Office of Public Affairs (ASC/PA) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

Brian Stucke

BRIAN STUCKE
Project Engineer

Patrick Price

PATRICK PRICE
Supervisor

Gerald Shumaker

GERALD SHUMAKER, Chief
Manufacturing & Engineering Systems Division
Manufacturing Technology Directorate

"If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify WL/MTII, Bldg. 653, 2977 P St., Suite 6, W-PAFB, OH 45433-7739 to help us maintain a current mailing list."

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	MAR 1994	FINAL

4. TITLE AND SUBTITLE
INTEGRATION TOOLKIT AND METHODS (ITKM), AUTOMATED PROTOCOL ANALYSIS/REFERENCE TOOL RESEARCH AND DEVELOPMENT PROJECT (APART)

5. FUNDING NUMBERS
C F33615-91-C-5711
PE 78011
PR 3095
TA 06
WU 41

6. AUTHOR(S)
FRANK J. WROBLEWSKI
J. DOUGLASS H. WHITEHEAD

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
CORPORATION FOR OPEN SYSTEMS INTERNATIONAL
8260 WILLOW OAKS CORPORATE DRIVE
SUITE 700
FAIRFAX VA 22031

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
MANUFACTURING TECHNOLOGY DIRECTORATE
WRIGHT LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT PATTERSON AFB OH 45433-7739

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
WL-TR-94-8014

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A research prototyping project to improve the data communications portion of enterprise integration. The project focuses on prototype experiments in the construction of a highly interactive graphics oriented tool to automate the design, construction, and implementation of data communications protocol. The project includes the building of an example application that uses protocol designed and generated with the APART tool. And finally, the project includes usability testing of the APART tool.

14. SUBJECT TERMS
DATA COMMUNICATION PROTOCOLS, VISUAL PROGRAMMING CASE TOOL, DISTRIBUTED APPLICATIONS, PROTOCOL VALIDATION & SIMULATION, ENTERPRISE INTEGRATION

15. NUMBER OF PAGES 89

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

List of Figures

1.0 Summary

This document reports on the results of the Automated Protocol Analysis/Reference Tool (APART) which is being built by the Corporation for Open Systems under contract with the United States Air Force under the Integration Toolkit and Methods (ITKM) project, Contract F33615-91-C-5711. This is a research prototyping project to improve the data communications portion of enterprise integration. The project includes prototype experiments in the construction of a highly interactive graphics oriented tool to automate the design, construction, and implementation of data communications protocol. The project includes the building of an example application that uses protocol designed and generated with the APART tool. Finally the project includes usability testing of the APART tool.

The Automated Protocol Analysis/Reference Tool (APART) is a protocol analysis CASE tool for the design, analysis, simulation and validation, and code generation of protocol, which can be used to develop communication systems using open system communication protocols and access methods in a UNIX environment. APART provides an easier, faster, more reliable method of developing protocol applications for use in enterprise integration. APART handles protocols that specify concurrent communication between two or more parties. Existing user applications and communicating entities which were implemented external to the tool environment can be integrated with protocol generated with APART.

Protocols are sets of rules that govern the interaction of concurrent processes in distributed systems. Designing a logically consistent protocol that is useful and deadlock free is a challenging and often frustrating task.

APART is a tool which allows a protocol designer to graphically define and manipulate a protocol at a high level of abstraction. Much of the design process is automated.

APART provides the protocol developer with a sophisticated graphics interface to efficiently and effectively employ flow diagrams to design *who* is to communicate, state transition diagrams to design *when*, and message format diagrams to express *what* is to be transmitted. These diagrams convey complex relationships and behavior of the protocol description to the designer.

There are three kinds of diagrams in which a protocol specification is built: Data Flow Diagrams, Finite State Machine (FSM) Diagrams, which include event/action diagram (notation), and ASN.1 Diagrams. The Data Flow Diagram describes the channels of communication, (i.e. the flow of messages) between individuals (FSMs). Each FSM in a Data Flow Diagram can be expanded into a state transition diagram, called the Finite State Machine Diagram. The FSM Diagram focuses on protocol states and how they are related to each other. A FSM has a start state and is stimulated to change state by the arrival of messages. Preconditions for transitions from one state to another as well as side-effects resulting from their occurrence are specified in the FSM diagram. One can wait on the arrival of a message, send a message to a peer, test or set a variable, invoke a "C" function, or even specify "C" code in line. The structure of the message and how the message is to be filled with data are further defined in the ASN.1 diagram and ASN.1 message constructors respectively. Provision is made for additional "C" code and libraries.

From an early stage in the design process, APART can simulate the protocol and employ validation techniques to uncover design flaws and incompatible protocol options in an interactive environment. When problems such as deadlock are detected, the developer can redesign the appropriate parts of the protocol and repeat simulation and validation until the design is complete. Multiple design passes minimize unforeseen problems, and reduce the time required to integrate data communication protocol and the user application.

When the protocol is complete and without known errors, APART generates the code for the protocol. The collection of diagrams is translated into a set of executable tasks with a single executable task produced for each finite state machine diagram.

The C-like source code generated for the distributed protocol utilizes open systems. APART provides hooks into the generated "C" code where the designer can insert "C"-function calls to perform application level tasks.

Many different disciplines, such as client/server systems, distributed applications, collaborative systems and communication protocols can benefit from the APART tool. Groupware is one such example. Groupware is the implementation of group behavior protocol, in that there are messages passed back and forth between computers which, in the context of available resources, enforce the rules which define the course of the interaction between the members of the group. Some examples of Groupware (collaborative distributed) applications such as, video teleconferencing, computer teleconferencing, business meetings, group authoring, RFP Development Process, RFP Evaluation Process, Research and Development, Business Planning and Budgeting are but a few.

2.0 Introduction

This document reports on the research pertaining to the Automated Protocol Analysis/Reference Tool (APART) which is being built by the Corporation for Open Systems under contract with the United States Air Force under the Integration Toolkit and Methods (ITKM) project, Contract F33615-91-C-5711. The APART tool allows a communications protocol designer to define and manipulate an Open Systems Interconnection protocol at a high level of abstraction. From this design APART will create a reference implementation of that protocol, and validate the protocol in an interactive environment with facilities to change the protocol as design flaws are uncovered.

2.1 Scope of This Document

This document defines in detail the technical items investigated as part of the research project. Its main purpose is to document the results of the research.

2.2 Overview of This Document

This section introduces the project, sets the scope of the document, identifies related documents, and lists terms which are used throughout the document.

Section 3 provides a brief description of the problem.

Section 4, describes the design goals of APART as they are laid out in this section. It provides a high-level description of the technologies involved, and describes the major components of APART.

Section 5 discusses motivation of APART design and project methodology.

Sections 6 covers the results of the investigation. It starts with a walk-through of the APART tool and how protocol is represented. It enumerates the features and functions that were the subject of the prototyping experiments. It includes a discussion of results.

Section 7 deals with APART tool applications. It talks about groupware, protocol independence as an application architecture and how APART can be used in this domain.

Section 8 reports on the major protocol application example, Collaborative Writing.

Section 9 reports on the usability testing.

The Appendix documents some initial protocol analysis that lead to the basic protocol building blocks within the APART tool.

2.3 Related Documents

The APART project was proposed to the Air Force in response to their notice in a Department of Commerce publication requesting submission of ideas for projects which would further the goals of Enterprise Integration.

The Corporation for Open Systems submission, dated August 14, 1990, is entitled, "The Automated Protocol Analysis/Reference Tool (APART) Research and Development Project Technical Proposal."

The contract covering the work on this project is described in "Integration Toolkit and Methods Contract F33615-91-C-5711."

Deliverables include a tool user manual titled "The ENGAGE Protocol Development Environment User's Guide.

2.4 Definition of Terms

In order to head off confusion in the rest of this document frequently used terms are defined below.

Full Protocol Stack - A collection of protocol implementations which cover each Layer of the ISO Reference Model for Open Systems Interconnection.

Protocol Stack - A collection of protocol implementations which cover a contiguous subset of layers of the ISO Reference Model.

Protocol Implementation - A module(s) of computer code which carries out the functionality of a single protocol.

Protocol - Refers to a collection of events, actions, states, and message format that pertain to a single layer, sub-layer or service element of the OSI Reference Model.

ISO Reference Model - Standardized hierarchy of protocol services.

Protocol State - A component of a means of describing protocol. A state is a form of memory, in that one will remain in the state until some pre-specified condition(s) occur. Any subsequent action can take into account the current state.

Finite state machine - A means to describe a protocol. A FSM is a set of states and transitions between states. In some sense, a state is a form of memory, in that one will remain in that state until some pre-specified condition, or conditions occur. Each condition is represented as a transition and they dictate both what action should be taken, and what new state to visit.

A Path through a FSM - is a sequence of [state, transition,...,state,transition], where each node and arc in the sequence is in the FSM.

AFSM - Augmented Finite State Machine is a means to describe a protocol in which the notion of states, events, and actions are combined with the notion of predicates or variables.

Full State Search Space - It is a single pure FSM constructed by flattening AFSMs for the communicating parties into pure FSMs, and combining them. A "full state" is a state in the full state search space.

Pure FSM - A FSM composed solely of states and transitions. No constructs such as variables or queues are used. This form of FSM lends itself to search algorithms useful in many forms of validation.

Deadlock - A failure of communication. It is exhibited when both ends of a channel are waiting indefinitely for the other side to do something.

Livelock - A weakness of communication. A protocol has a livelock, if it allows endless interaction between to FSMs, without progress.

Unreachable States - A state is said to be "unreachable," (from the start state) if there is no path from the start state, to that state.

3.0 Problem Description

Enterprise integration is an umbrella term that covers all technologies that will assist in merging all levels of computer based work into a single environment. Data communication enables enterprise integration. To further this end, a major industrial thrust to improve computer protocol options and services is the development of open system network communications standards. The open systems process has achieved some significant milestones such as X.400, electronic mail. However, problems of integration and deployment remain, and are only heightened by the acceleration of technology. Tools must be devised that handle the explosion of protocol details and provide meaningful and timely information to the user.

3.1 Enterprise Integration and Benefit of Open Systems

In many corporations, different organizational elements within the enterprise seek out their own system solutions for their requirements, thus creating "islands of automation" and "islands of data". Often their systems are built upon different vendors' proprietary solutions. Communication between these islands is difficult and expensive because of the multi-vendor nature of their systems environment. This difficulty inhibits the flow of critical information throughout the enterprise.

One of the keys to enterprise integration involves adopting a common standard for communication. Some of the advantages of using open systems network standards include:

- Allows products from multiple vendors to communicate, giving the purchaser more flexibility in equipment selection and use.
- Enables multi-vendor interoperable networking environment.
- Increases access to information throughout the enterprise.
- Reduces life cycle cost of maintaining and upgrading systems.

The open systems communications model most widely accepted around the world is the Open Systems Interconnection (OSI) Reference Model and TCP/IP for data communications networks. The goal of the seven layer model is the realization of interoperable systems based on internationally accepted data communications standards. The basic objective of the OSI and TCP/IP models are to enable the interconnection of a multi-vendor networking environment that supports a wide range of applications. Each layer of the model defines a set of functions for supporting communications across a multi-vendor system environment. The model provides a framework for the development of standardized protocols and products for achieving enterprise integration.

To date, many standard protocol specifications have been developed based on this reference model. New protocols are being devised for more and more aspects of enterprise activity, to meet the ever growing demands for automation within the enterprise. As the design and integration of such protocols accelerates it becomes imperative that automation tools be applied in order to handle the explosion of protocol details available in a standard, while maintaining quality of service. Our research focuses on tools to effectively handle the quality and complexity of protocol information.

4.0 APART Tool Description

The previous section of this document discussed the need for an automated tool for design of such protocols. The current section will provide a high-level description of the APART tool's design. It attempts to paint with broad brush strokes a picture of the tool as a whole, giving the reader an appreciation for the overall system architecture and the capabilities and features of the tool without losing the reader in a sea of intricate details. It serves as an introduction to the later sections of this document which provide more comprehensive treatment of the major design components of the APART tool.

APART may be viewed as consisting of several major modules depicted in Figure 1:

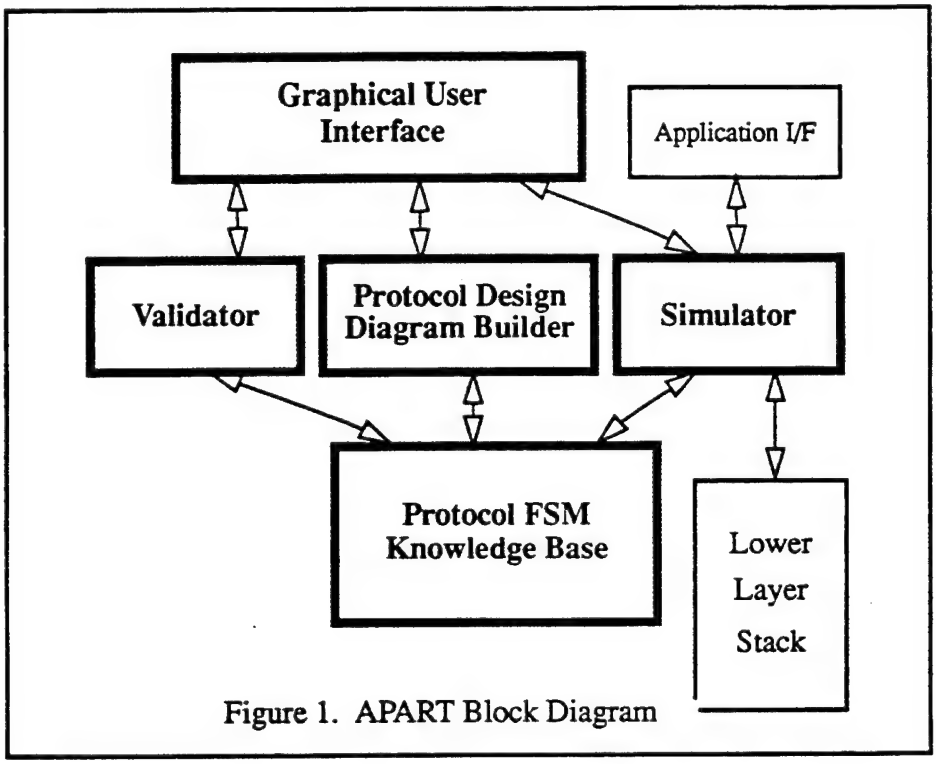


Figure 1. APART Block Diagram

1. Diagram Builder - This module supports the underlying functionality of APART for design. It constructs and organizes the underlying objects that will be used throughout the APART tool.
2. Simulation module - This module supports the simulation of protocol activity.
3. The protocol validation module - This module automates the detection of design flaws in the protocol being designed.
4. A graphical interface module - Provides graphical, direct manipulation interface to the functionality provided by the other major modules.
5. Knowledge Base - This module stores and retrieves protocol components.

6. Application Interface - Supports the attachment of an external user application to APART tool protocol reference implementations.
7. Lower Layer Stack - Supports the transmission of a message, protocol data units, to an external entity outside the APART tool environment.

Below is a discussion of the design goals for the APART tool. Following this is a description of the model of the design problem supported by APART. Next will follow an overview of the major modules that comprise APART. This section concludes with a description of the hardware and software environment that will be used in implementing and using the APART tool.

4.1 Design Goals in Brief

The over-riding goal of this project is to build a protocol design tool which can deal with the complexity of open systems protocols and thereby provide an easier, faster, more reliable method of developing protocols for use in enterprise integration. To achieve this overall goal the design tool must provide several specific capabilities to the designer:

First, it must allow the designer to manipulate a communications protocol at various levels of abstraction. This should improve user understanding of the protocol by providing information at the most advantageous level of detail as dictated by the understanding and needs of the user. As such, the user is more likely to choose the correct service options, and thereby reduce the time to successful inter-communication.

Second, the tool will employ graphics in order to more efficiently and effectively convey protocol description and behavior to the user. Graphics has a much larger bandwidth to convey complex relationships to people. As such, the graphical display of information reduces the time required for a user to absorb the information, which in turn will speed the design process.

Another design goal is to automate as much of the design process as possible. This will make it feasible to try alternative protocol designs before committing to a final set of protocol options.

APART will employ protocol validation techniques to uncover design flaws and incompatible options earlier in the design process. Again, this will reduce the time required to integrate data communication protocol.

Finally, APART will automate the process of going from design to implementation. The final output of the APART tool is a reference implementation of the newly defined protocol. This should reduce the likelihood of introducing errors by reducing manual translation.

4.2 Model of the Design Process Supported by APART

This section will describe the conceptual model of the protocol design process which APART will convey to the user. The communications process may be viewed as multiple communicating entities, with each being supported by an open system protocol stack. Each stack is composed of layers and sub-layers of cooperating protocols that combine to allow communication with other stacks. Figure 2 shows a visual representation of multiple communicating entities.

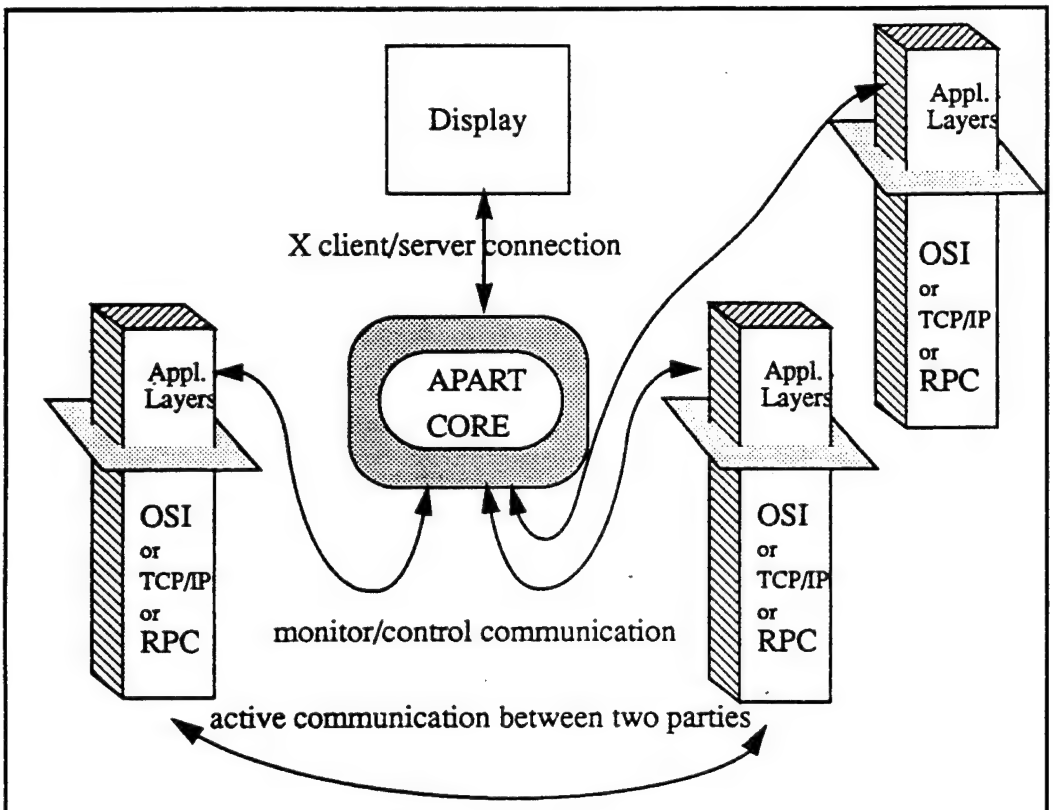


Figure 2. Protocol Simulation Communication Architecture

The APART Tool design environment can support protocols at any layer. APART's prototype code generator directly supports design and implementation of protocols at the application layer.

APART represents protocol as augmented finite state machines (FSM) - augmented in that APART provides predicates and variables which store state information (This is in keeping with the tradition of ISO protocol standards). The protocol designer defines the states of the protocol's state machine and for each state the events of interest to that state, the actions which are to be performed when one of the events occur, and the subsequent transition to another state.

Each machine has input and output queues which are used to communicate with the FSM above and the FSM below. The APART user defines the connection between the output queue of one FSM and the input queue of the next. Messages are transmitted between FSMs by placing them in the appropriate output queue of a protocol machine. ISO standards define service primitives for communicating between adjacent layers. The primitives pass parameter information as well as PDUs between layers. The ISO standards define the PDU formats using ASN.1 grammar, but do not specify the format of the parameters. In the APART environment, all communication between layers is via messages which include parameters and PDUs. The complete format of all messages will be defined in ASN.1 like notation.

APART provides two modes of operation:

1. Simulation Mode - allows the user to check a protocol by single-stepping through a message transmission sequence and examining internal variables at each step.
2. Reference Implementation Mode - The multiple executable tasks, reference implementation, are completely independent from the APART environment.

4.3 An Object-Oriented Tool

APART is designed as an object-oriented tool for protocol design. The user manipulates objects in order to perform work. The object-oriented paradigm is appealing for this application for several reasons. The APART tool models real-world entities - stacks, protocol machines, states, etc. It seems natural to define objects to represent them and an interface which provides graphical manipulation of these objects as a means of defining, examining, and changing them.

The object-oriented paradigm also facilitates separation of the internal functionality of APART from the user interface aspects of the tool. The de-coupling of function from display allows changes in one module without affecting the other. The user interface may change substantially without changing the underlying functionality. Alternatively, significant changes can be made in the core design and simulation functional objects without affecting the user interface.

This de-coupling of underlying functionality from display also facilitates flexibility in the level of detail which is displayed. If each underlying object has a default display image, then the user can peruse the protocol design and the communications process at various levels of detail -first viewing information about the protocol as a whole, then narrowing the focus to a particular state, then further narrowing focus down to one event for that state, and finally examining at different levels of detail some of the actions to be performed when that event occurs.

An object model of the protocol design process also fits well with a direct manipulation interface for the definition and manipulation of a protocol. During the protocol definition process, the user selects an object which represents part of the definition of the protocol and sees a display which allows definition or editing of that object, thereby defining that part of the protocol. The part may in turn be composed of other objects and selecting one of them may call up a display of that object's component parts.

4.4 High Level Design

4.4.1 Core Design and Simulation Modules

The Design and Simulation modules (see figure 1) provides the core functionality required to design and edit a protocol, test it under various conditions, examine its component parts at various levels of detail, and support connection to and communication with communicating entities defined and implemented external to the APART environment. These modules will be implemented as a set of objects corresponding to the components of the communications process.

There is a global SIMULATOR object which maintains information which is global - e.g. the mode of the simulation, the entities which are communicating, etc. The SIMULATOR maintains a list of STACK objects, one for each communicating entity, and performs coordination of execution between stacks so that there is an orderly communications process.

The STACK object in turn maintains a list of protocol machines which connect an output queue of one FSM with the input queue of the next.

Each FSM of a stack is implemented as a PROTOCOL MACHINE object. A PROTOCOL MACHINE maintains a list of execution states, the transitions between states, and a set of input and output queues by which the machine communicates with the FSM above and the FSM below.

Each TRANSITION object is composed of a set of events and the action scripts which are executed when one of the events occurs.

A MESSAGE object defines the ASN.1 format of a message and may contain the actual data content of an incoming or outgoing message.

Side-effect objects define behavior for checking the contents and syntax of a message and for manipulating pieces of a message in construction of a new message. Side-effect objects defines particular kinds of action to be performed. Examples of actions are:

- variable testing and manipulation
- message checking
- message manipulation
- message transmission
- external function calling

As part of the object-oriented nature of APART, it maintains libraries of re-usable components of entire protocols or pieces of protocols which can be used and modified to create new protocols. Among the useful components which may be stored are:

- protocol machines
- message formats (ASN.1 definition of the syntax of a message)
- entire protocol stacks

4.4.2 Protocol Validation Module

The Protocol Validation module will test possible state combinations between communicating peers. It will perform reachability analysis on the protocol machine to verify that

states may be reached and that no deadlocks will occur. This is a very costly APART tool activity in terms of CPU time and memory space due to the combinatorial explosion of the search space which occurs when all possible execution paths are considered. APART will try to bound the problem by identifying combinations of states between the two peer machines which are nonsensical. By doing this, the size of the problem and the time required to perform the required search should be significantly reduced.

4.4.3 Graphical Interface Module

APART is a software tool that a protocol designer will use on a workstation with a high resolution monitor, keyboard and mouse. The software will display a graphical depiction of the communications process being modelled. Components of a communications protocol are defined in APART via graphical manipulation of objects that represent the corresponding components. The design programming model in section 6.1 gives an introduction to APART graphics by presenting three examples.

4.4.4 Application Interface

The APART tool provides a communication service to a user application. Plans calls for such an interface to drive the programming interface of the FrameMaker[1] product.

4.4.5 Lower Layer Stack

For this version of APART the lower layer stack is defined to start with the DSET[2] application programming interface (API). This is an application layer interface. The communications model is based on OSI Association Control Service Element (ACSE) and Remote Operations Service Element (ROSE) international standards. The same Communications Infrastructure API is provided on top of a variety of protocol stacks, including TCP/IP, and OSI.

4.5 Implementation and Operational Environment

APART is implemented on a SPARC 2 running Sun OS (Unix) and X-Windows. It is implemented in Common Lisp.

The object model, i.e. the group of underlying objects which implement the APART tool functional module, is implemented in Garnet[3].

The graphical interface for APART is implemented in Garnet, a graphical interface toolkit (written in Common Lisp) from Carnegie Mellon University.

Lower layer support is provided by the Distributed Systems Generator (DSG) Tool Kit[2].

Initial development was done on Sun 3/60 workstations. Equipment upgrade to SPARC 2s was accomplished during the project.

---

1. Frame Technology Corporation
2. Distributed Software Engineering Tool (DSET) Corp.
3. School of Computer Science at Carnegie Mellon University

5.0 Design Goals for APART

The overall goal was to prototype a tool that can deal with the complexity of open systems protocols and thus improve the speed and quality of enterprise integration. To achieve this overall goal; APART, in concert with a user, should be able to:

1. Manipulate data communication protocol at various levels of abstraction- This should improve user's understanding by providing information at the most appropriate level of detail, as indicated by the understanding and needs of the user. By enhancing the user's understanding of a protocol, the user is more likely to choose the correct options. This will speed integration.
2. Employ graphics in order to more efficiently and effectively convey protocol description and behavior to the user- Graphics have a much larger bandwidth to convey complex relationships to humans.
3. Automate protocol validation- This should reduce the integration time by identifying protocol design flaws or incompatible options earlier.
4. Automate the production of an implementation design- This should reduce the likelihood of introducing errors by reducing the manual translation.

Through the above automation goals, make it feasible and cost effective to try alternatives before committing to a final set of protocol options. This should improve the likelihood of selecting a set of protocol options with the desired behavior in a shorter amount of time.

5.1 Technical Motivational Overview (Assumptions)

The vision of the APART development staff has been to construct a workstation environment that will maximize the productivity of a protocol designer and integrator.

There are three categories of ideas that have contributed to the design of APART:

1. **The Semantics of Protocol** - There is a distinct advantage of having a protocol in machine processible form that is amenable to viewing, checking, and cross checking from various points of view. The process of encoding a protocol in a form understandable by a machine requires an unambiguous discipline. This can result in a protocol and its specification that is more exactly understood than the current norm of protocol standards.
2. Some technologies that are relevant to the manipulation of protocol design:

   Protocol Simulation - One might want to "try out" a protocol before work begins on an efficient implementation.

   Protocol Validation - An automated check to catch mistakes.

   Iterative Design - By allowing multiple design passes, one can minimize unforeseen problems.
3. Technologies indirectly relevant to protocol design, but facilitate productivity:

   Rapid Prototyping - A design philosophy centered around the assumption that by minimizing the labor in design construction, one encourages iterative design.

GUI - Graphic User Interfaces can be an effective Human Computer Interface (HCI) technique. By relaying information visually, often a human can track more details than text based approaches.

User Sensitive Information Filtering - By allowing the protocol designer to specify what is viewed, one can avoid confusing detail.

Software Reuse - Object oriented design will allow the construction of libraries. By storing/using protocol as modular chunks, one can reuse old pieces of protocol.

APART integrated these technologies into a single tool. In so doing, our hope has been to overcome the productivity encumbrances experienced when integrating open systems protocols as part of enterprise integration.

5.2 The APART Project (Methods)

APART is a rapid prototyping project. The research method is to prototype the ideas specified in the design document. Opportunities for feedback are expected in this process. The first will occur when the researchers themselves use APART to encode the example protocols. Design modification and retry is a normal step in the prototyping method. Once the researchers have evolved the prototype to a working level other users will be introduced to APART. This will provide a second level of feedback that once again will be used to enhance the design. This final report documents the discoveries learned during the prototyping process.

A groupware application referred to as collaborative writing has been chosen as the major example application. Collaborative writing is a distributed client server application designed and generated by use of the APART Tool.

APART has several major characteristics:

1. Provides a design framework for constructing protocol implementations.
2. Performs simulation based on the protocol specification entered.
3. Manipulates protocol at multiple levels of abstraction.
4. Provides entry and display of protocol specification via a graphics interface.
5. Performs protocol validation checks.
6. Provides a second mode of operation in which the encoded protocol knowledge behaves as an implementation.
7. Provides connectivity to an application entity.
8. Provides connectivity to a data communication service.

The last two items are provided for the example protocols that are implemented using APART. This is for example only, APART is intended as a general tool that can be used to define any open system protocol at any level. Protocols thus defined can be stored in a library and used as reference implementations, or as building blocks for other related protocols.

6.0 Results

In order to meet our objectives, there was a need to select or invent some sort of language that would allow one to develop protocol. The model had to be rich enough to express real world protocols and simple enough to be easily understood by protocol engineers. Rather than invent our own programming model, the team chose to adopt one widely used in the protocol and standards community. The choice was finite state machines, augmented with variables and queues.

To establish a basis for this research, the programming model had to be rich enough to express real world protocol. One has to be able to define messages passing between two or more parties. One had to be able to define when a message is to be sent, and where it is to go. When a message arrives, one must be able to define what to do with it. If these characteristics exist, then the programming model is rich enough.

The augmented finite state machine model is rich and well understood. Considerable literature in protocol validation use this model. In fact, most protocol standards use an augmented finite state machine as a basis for description. Indeed, many problems of standards interpretation result from deviations from the AFSM model through textual descriptions. So the team felt confident that augmented finite state machines were expressive enough to design protocol. Another model that could have been used is the Petri Net design model. This was a strong possibility, as it is commonly found in protocol research literature.

Ideally, the programming model should also be easily understood by protocol engineers. The team felt that engineers are accustomed to thinking and expressing protocol in terms of states, queues, and variables. So, rather than force an engineer to learn a new language, the team chose to reinforce existing notions of protocol by displaying states graphically. Finite state machine diagrams are commonplace in protocol literature, and they allow one to "see" the relationships between states.

The team chose interfaces for the design of protocol from other disciplines as well. Data flow diagrams were adopted from the software engineering field. They define the delivery paths of messages, which emphasize "who is talking to whom", rather than mechanical construction and use of an address. The ASN.1 diagrams that display message content, were adopted from the visual presentation of structure technique called "Tree-Maps".

By supporting the protocol programming model with a graphical look and feel, the team has met programming model requirements, expressiveness and ease of use.

6.1 Design Programming Model

The design programming model will be presented by walking through three protocol examples. A simple client/server protocol, "fsm-timer", will comprise the first example. This protocol is made up of two communicating entities. The protocol includes one advanced feature that enables it to support many instances of the client entity concurrently. This feature demonstrates a many-to-one relationship. The second example, "pool", is made up of three types of communicating entities. It includes many-to-one and one-to-many relationships which on aggregate demonstrate how a many-to-many relationship can be supported. The final example will combined the first two examples, demonstrating the concept of protocol reuse.

Describing the APART environment centers around the interactive graphical diagrams. The following window group, Figure 3, provides a composite view of all the diagrams used to define a protocol. Each diagram will be defined as part of the walk-through.

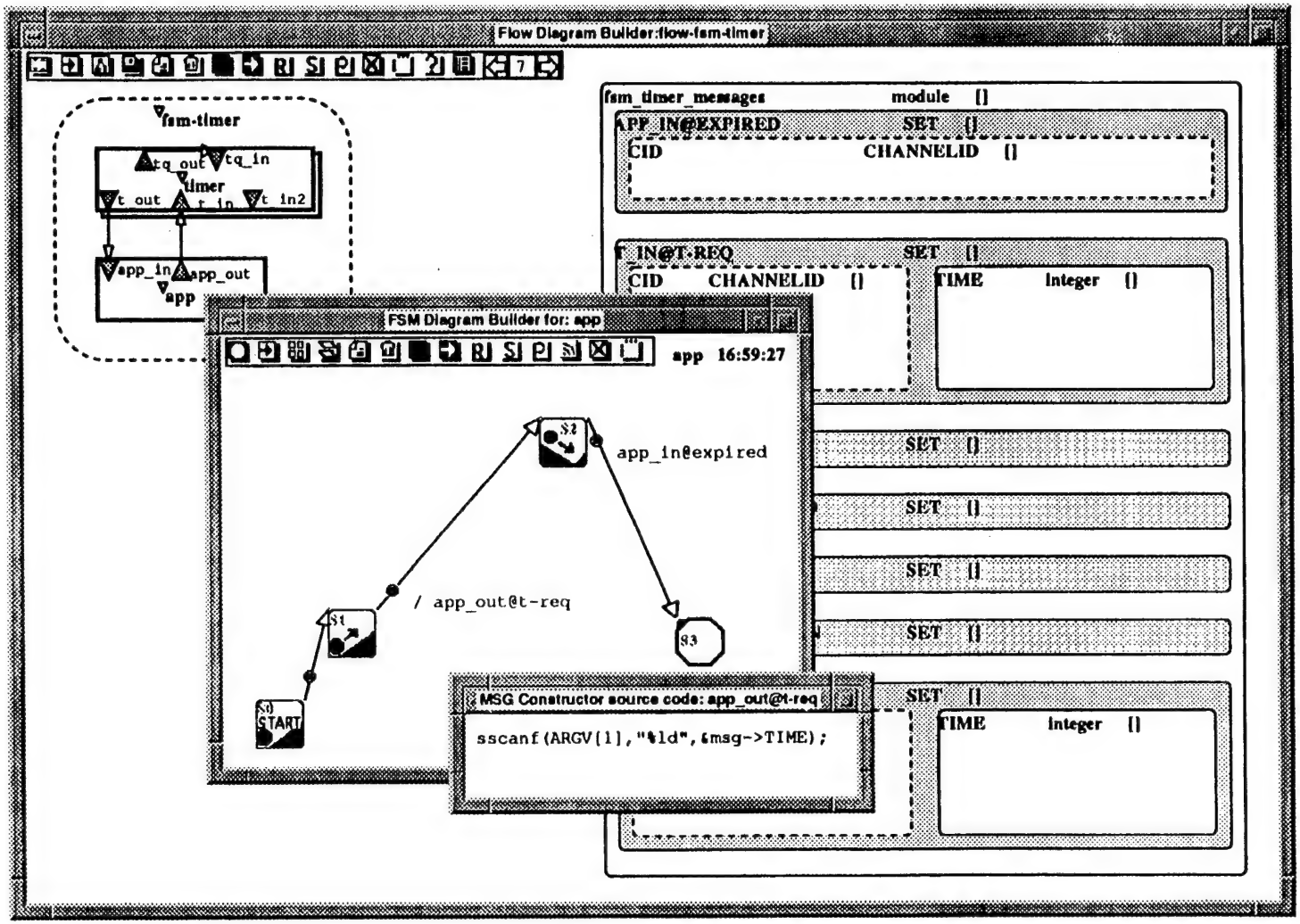


Figure 3. APART Window Composite View

6.1.1 Flow Diagram

The first step in protocol design is to define the high level organization. The flow diagram is used to accomplish this task. Figure 4 contains a flow diagram for the client/server protocol.

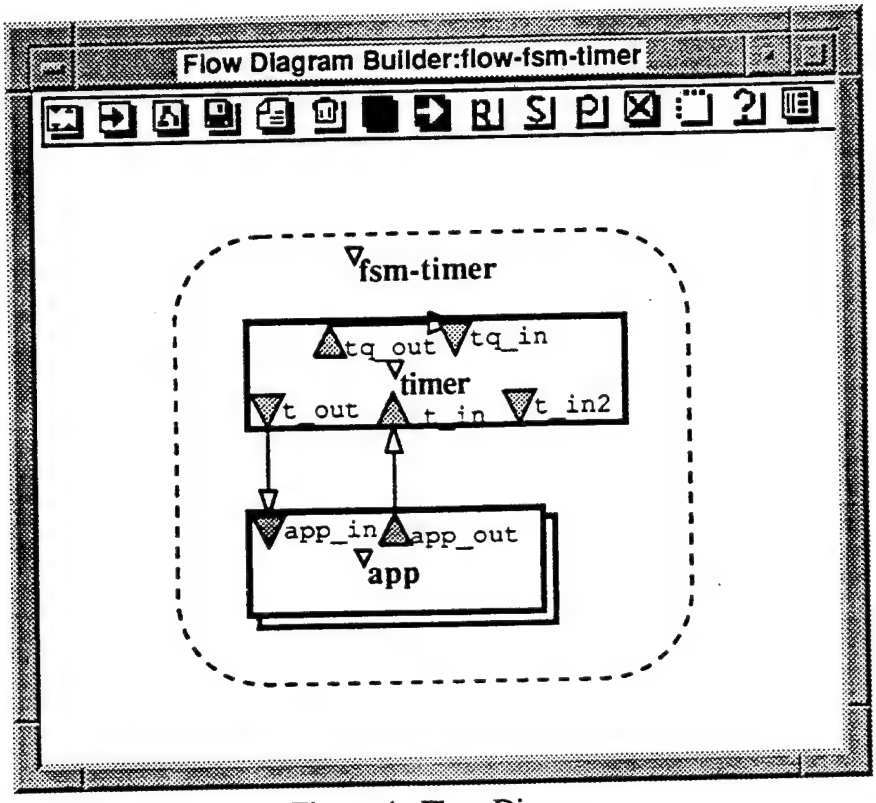


Figure 4. Flow Diagram

The diagram is made up of rectangles which represent the communicating entities. They are referred to as finite state machines (FSMs) because of their internal structure. This example contains two FSMs. The server FSM is titled "timer" and the client FSM is titled "app". The server provides a timekeeping service. The client can send a message to the server requesting that a message be sent back after a specified amount of time has elapsed.

Channels are defined to be objects that can send or receive messages. Channels are represented as triangles in FSMs. Output channels are connected to input channels using arrows. The arrows are referred to as connections. Channels and connections make up unidirectional paths over which messages can flow. In the above window the "app_out" channel is connected to the "t_in" channel, and the "t_out" channel is connected to the "app_in" channel.

The two connections provide a bi-directional path for clients and servers to communicate with each other. In some cases it is advantageous to be able to send a message to yourself. In our example there are two additional channels, "t_in2" and "tq_out", which allows the "timer" FSM to put messages in its own input queues.

The usage of these channels will be covered later under FSM diagrams. The FSMs of a protocol can be packaged into a single entity referred to as a stack. A stack is represented by the broken line rectangle and all the objects that are within it. A stack is an organizational unit that can be saved on disk. This covers all the parts of the "fsm-timer" data flow diagram. Data flow diagrams contain the "who is communicating with who" information.

6.1.2 FSM Diagram and State Transitions

The next step in the design process is to define the interaction between FSMs. The interaction is made up of the exchange of messages under predefined conditions. The conditions are sometimes referred to as the rules of the protocol. To see the interaction each FSM can be expanded into an FSM diagram. APART uses an FSM diagram to represent the protocol rules. Figure 5 contains the server FSM diagram, "timer".

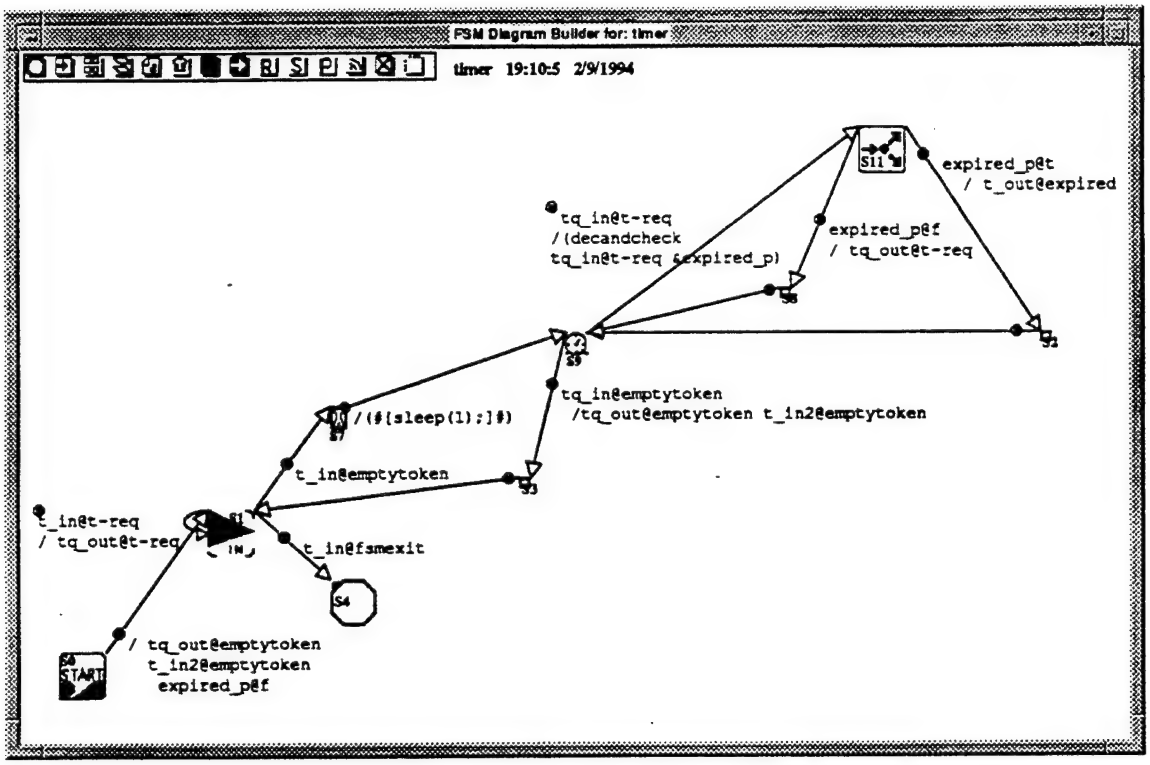


Figure 5. FSM Diagram

The FSM diagram is made up of states represented as icons and transitions represented as arrows. The diagram is interpreted by starting at the START state and following the arrows. The combination of states and arrows make up paths in the protocol. All the combinations of paths define the protocol behavior, rules.

The "timer" FSM task will accept incoming messages. The messages will contain an integer value which represents the amount of elapsed time before a return message should be sent. The "timer" will decrement the value as each second passes. When the count reaches zero it will send the "expired" message to the client.

To be able to handle a variable number of requests a queue is used. A queue can be constructed out of two channels that are wired together. The data flow diagram shows that the "timer" has two channels, tq_out and tq_in which are wired together. All channels can be thought of as queues. The term 'hopper' is used as an abbreviation for the special case where the in and out channels are of the same FSM.

Each transition can have one or more preconditions which must be true before the path to the next state can be taken. Each transition can have one or more side-effects. If the path is taken then the side-effect actions are performed. Side-effect actions constitute sending messages, setting variable values and executing program functions.

To read the FSM diagram you will need to interpret the preconditions and side-effects attached to each transition. Syntax for preconditions and side-effects are as follows:

1. preconditions / side-effects
2. preconditions ::= precondition <space> precondition | precondition
3. precondition ::= channel_name@message_name | variable_name@value
4. side-effects ::= side-effect <space> side-effect | side-effect
5. side-effect ::= channel_name@message_name | variable_name@value
   | (function_name param_1 ... param_n)
   | (chan_name@msg_name (constructor_function p_1 ... p_n))
   | (#[<excape to "C" source code]#)

The "timer" FSM starts out by putting an "emptytoken" message in the "tq_out" channel and the "t_in2" channel. Channels can hold messages and therefore can be thought of as message queues. The FSM then moves to the state.

This state has three different transitions that are possible. If it receives a "t_in@t-req" message it will put the message in the hopper and return to the same state. If it receives a t_in@fsmexit" message it will go to the "stop" state and terminate. If is receives a "t_in@emptytoken" it will move to the state.

Sooner or later this transition is guaranteed since the FSM put an emptytoken message in at the start state. The state unconditionally executes a side-effect funciton which put the task to sleep for one second, after which it moves to the state. This state checks to see if there are any messages in the hopper. If there is a message a side-effect function "decandcheck" will decrement the time counter in the message and set a variable which can be tests.

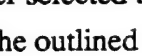
This brings the FSM to state S11, . S11 tests the "expired_p" variable. If time has not expired then the message is placed back in the hopper. If the time has expired then the message is used to construct a output message that is send back to the original client. The original message is then discarded. Either condition will lead back to state S9, . All the messages in the hopper will be processed once.

At this point, the emptytoken will have bubbled up to the top of the hopper. Arrival of the tq_in@emptytoken will take the FSM to state S3 and then back to state S1. All paths of the FSM diagram have now been traversed. It can be observed that a variable number of timer requests can be made and serviced with this diagram.

Its worth noting that the clients need not be aware of the internals of the server. As long as the clients adhere to the message exchange defined, the protocol will behave properly. This is in object oriented style. Different implementations of the timer can be substituted without having to modify the clients.

6.1.3 Message Diagrams (ASN.1)

Messages are typically made up of primitive components such as integers, character strings, and octet strings. The primitive items can be grouped together into sets or sequences. Figure 6 shows the messages for the "fsm-timer" protocol. This diagram shows seven messages that were generated from information in the FSM diagram. Most of this diagram was generated automatically by APART. Once the diagram was generated the user added the "TIME" field to the "t_req" message.

The "CID" fields were added automatically in support of the many-to-one relationship. This is part of APARTs default behavior in support of addressing. Note that this field is optional and can be overridden. The user selected the many-to-one option from a menu while constructing the flow diagram. The outlined FSM rectangle, 

, in the flow diagram is the visual feedback for this option being selected.

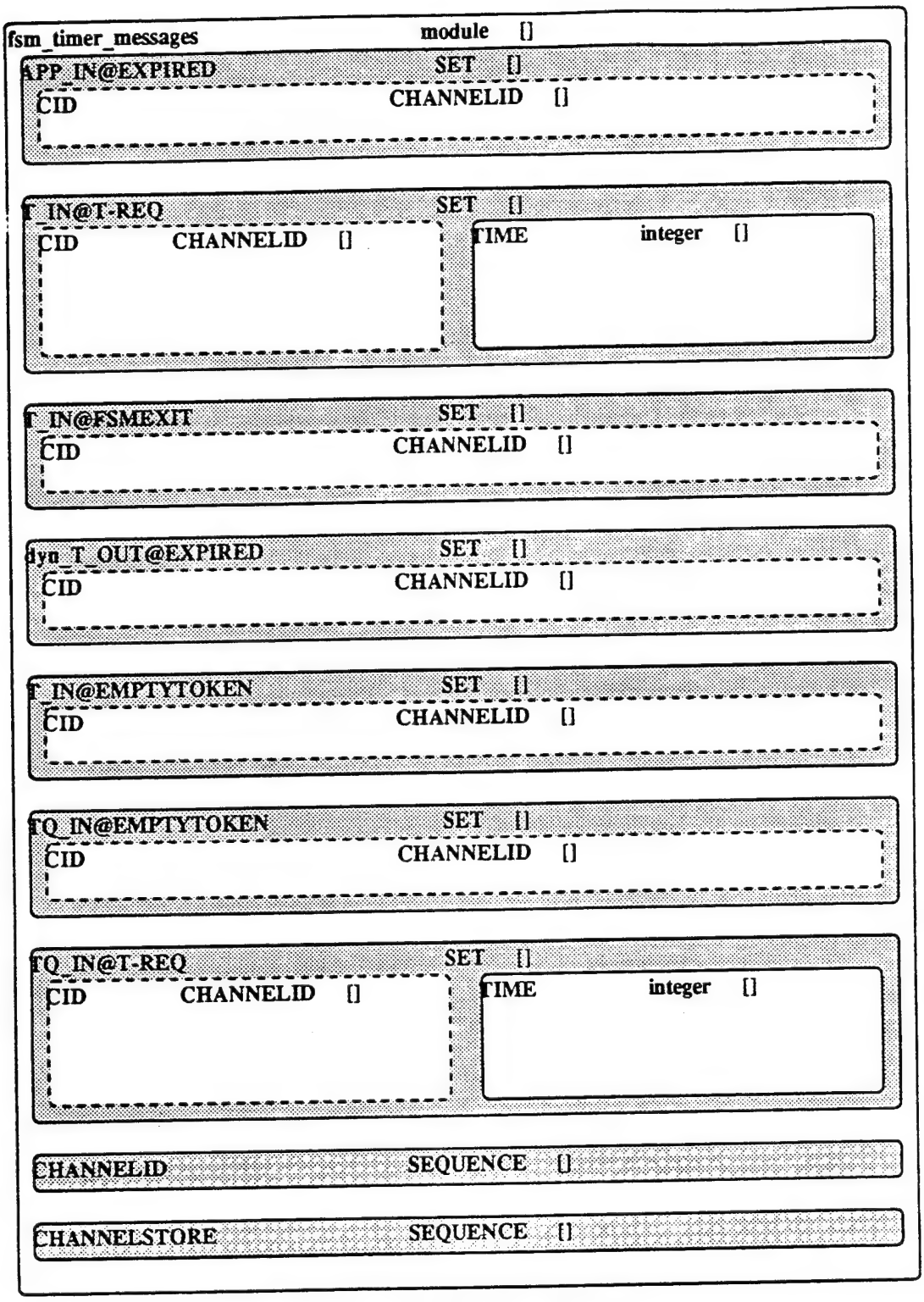


Figure 6. Message Diagram

When messages are put in the hopper a constructor function is used. In this case the message contents is simply copied. The constructor function is shown in Figure 7.

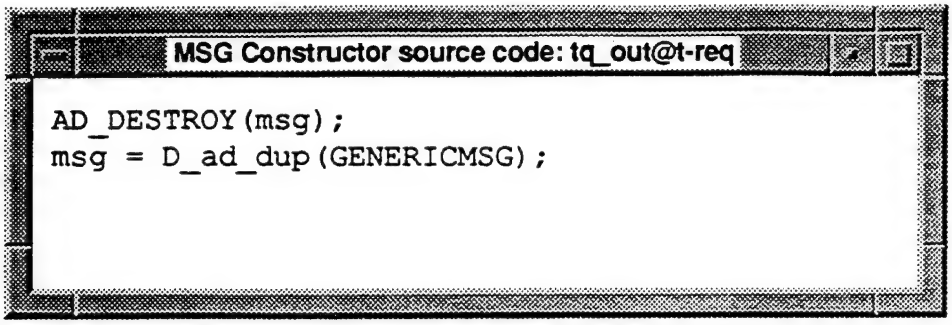


Figure 7. Constructor Function

At the time the constructor is activated a message buffer would have already been allocated. The typical scenario would be to fill in fields of the message. In this case the objective is simply to duplicate the message received. One way to accomplish this task is to delete the current message under construction. This is done in line one of the constructor. Line two of the constructor duplicates the last message received and assigns it to variable "msg". "msg" is a reserved word which points to the current message under construction. When the constructor is existed default behavior will send the "msg" message.

6.1.4 Side-effect Functions &Stub Files

The last piece of protocol definition to look at is the side-effect function "decandcheck". Source code for side-effect functions are stored in stub files as seen in Figure 8.

```
/*****************************************************************
 * timer.stubs.ac - Fill out the stubs in this file
 *****************************************************************/
#define lmsg GENERICMSG
/*
 * decandcheck, which is invoked by the following:
* decandcheck( (TQ_INqT_REQ *) evP->u.iv.adP, &EXPIRED_P );
 */
int decandcheck( TQ_INqT_REQp, EXPIRED_P )
TQ_INqT_REQ* TQ_INqT_REQp;
enum type_EXPIRED_P* EXPIRED_P; {
#ifdef DBG
 printf(" decandcheck: Enter\n");
#endif /* DBG */
 if (TQ_INqT_REQp->TIME > 0)
 {
 /* TQ_INqT_REQp->TIME = TQ_INqT_REQp->TIME - 2; */
 ((TQ_INqT_REQ*)lmsg)->TIME = ((TQ_INqT_REQ*)lmsg)->TIME - 2;
 *EXPIRED_P = EXPIRED_PqF;
 }
 else
 *EXPIRED_P = EXPIRED_PqT;
#ifdef DBG
 printf(" decandcheck: Exit\n");
#endif /* DBG */
 }
#undef lmsg
```

Figure 8. Stubs File

Most of this function was produced automatically as part of the code generation process. The "if" statement in the body of the function was the portion added by the user. The "if" statement both decrements the counter and sets the "expired_p" variable to "T" or "F". Constructors can access messages by use of reserved words, lmsg & msg, or by parameter passing. The comment in the "if" statement shows the alternate parameter method.

The Client FSM and its constructor is shown in Figure 9. The Client FSM "app" is very simple. From the start state S0, it unconditionally moves to S1. S1 unconditionally sends the "t_req" message to the server. Just before sending the message a one line constructor function takes an integer from the command line and places it in the "TIME" field. "app" then moves to state S2 where is waits for the" expired" message. Upon receiving the "expired" message it terminates.

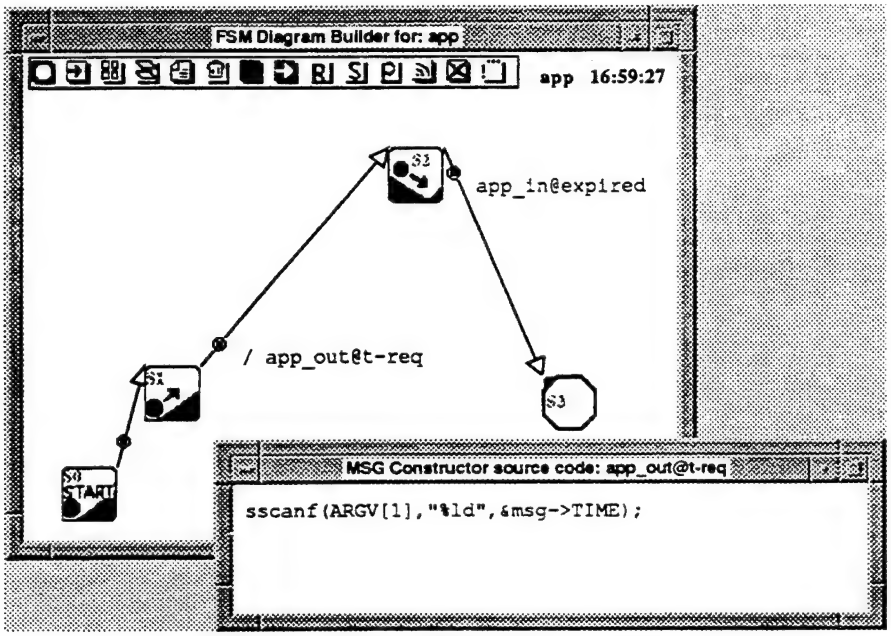


Figure 9. Client FSM and Constructor

All the building blocks used to define protocol in the APART environment have now been covered. This concludes the walk-through of the first example.

6.1.5 Work Flow Protocol Example

The second example is intended to show how robust protocol can be constructed out of the basic building blocks. Figure 10 contains a flow diagram for the "work pool" example. The "work pool" shows how a protocol can model business activity of a company. A fairly typical business activity is to match up customers who are requesting a service with employees, workers, who will provide the requested service. The key feature of this activity is that there are a variable number of requestors and a variable number of workers.

It is assumed that any requestor can be serviced by any available worker. For example purposes the protocol exchange will be the subject of focus and less on business activity. In this regard side-effect activity will not be shown. Typical side-effect activities would include database inquiry, calculations, and report generation of whatever the business deals with.

This protocol is defined as three finite state machines. One represents a worker, one represents a requestor and a third to manage the activity of associating requestors to workers. Initially requestors are not aware of workers. This means that requestors do not know the

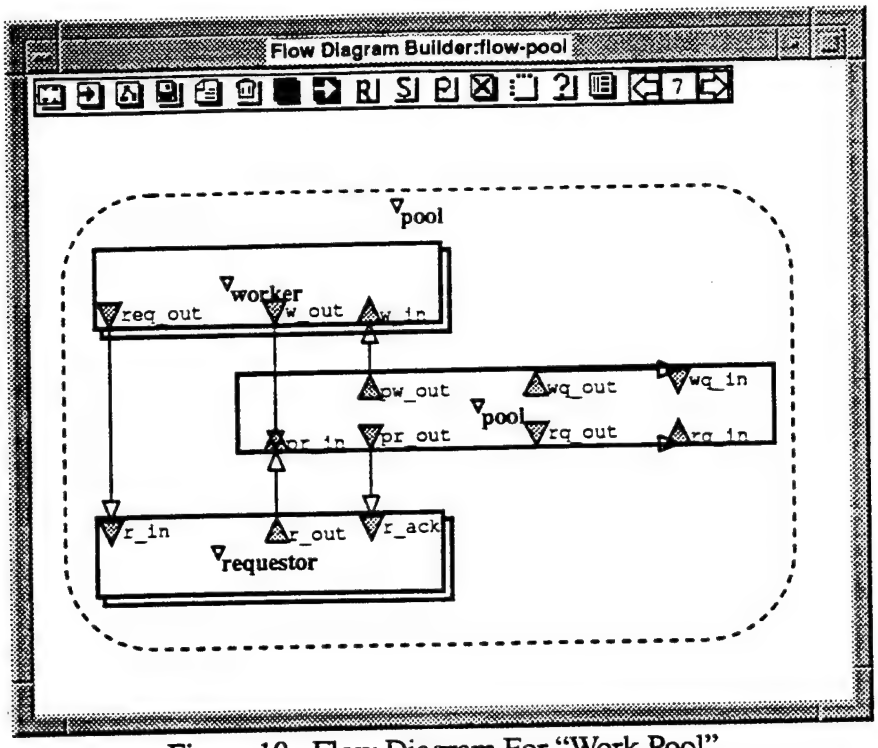


Figure 10. Flow Diagram For "Work Pool"

address of any worker. In this example there is only one pool FSM. At code generation time a unique identifier is allocated to the pool FSM. All worker and requestor FSMs know the pool FSMs identifier. Identifiers are converted to real network addresses at run time by a name server.

In the pool protocol four scenarios exist:

1. A worker announces availability to service a requestor, but there are no outstanding requests.
2. A request is made, but there are no available workers.
3. A worker announces availability to service a requestor and one or more requests are pending.
4. A request is made and one or more workers are available.

It is the pool manager finite state machines job to handle scenarios listed while remembering worker availability and requestor requests. As soon as there is an available worker and a requestor the pool manager will send a message to the worker identifying a requestor. Once the worker has received the requestors identifier, it can then communicate directly without the aid of the pool FSM.

Each of the FSM diagrams for the protocol will now be presented. Figure 11 shows the worker FSM.

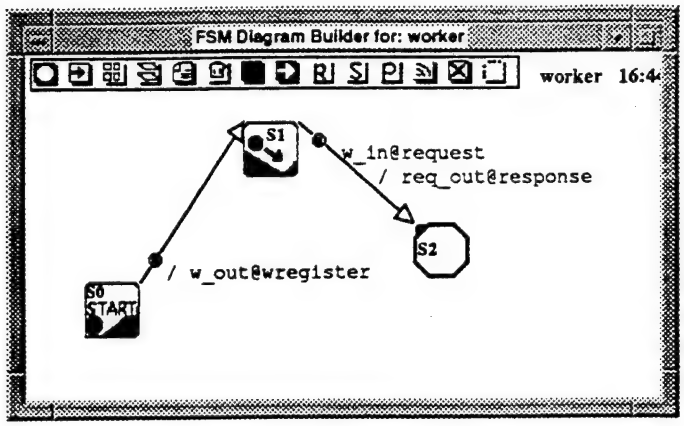


Figure 11. Worker FSM

Starting at the start state, the worker sends a "register" message to the pool FSM. It then moves to state S1. In S1 it waits for the pool FSM to pass on a "request" message from a requestor FSM. The request message contains the address of the requestor. The worker FSM can then use the address to construct a "response" message and send it directly to the requestor.

Figure 12 contains the requestor FSM diagram. Starting at the start state, the requestor

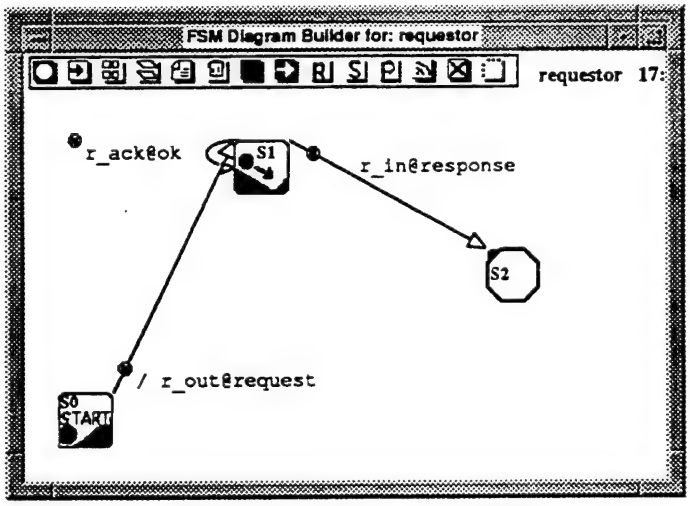


Figure 12. Requestor FSM

sends a "request" message. It then moves to state S1. Two messages are possible in state S1. It can receive an "ok" message from the pool FSM in which case it discards the message and returns to S1. It can receive a "response" message in which case is move to S2 and terminates.

Both these diagrams convey the simplicity of the exchange from a conceptual point of view. There is fairly sophisticated data communications activity taking place in support of

these protocol design concepts. This is the essence of APART tool usage. The protocol designer can concentrate on protocol design at a high level.

The next diagram to look at is the Pool FSM, see Figure 13.

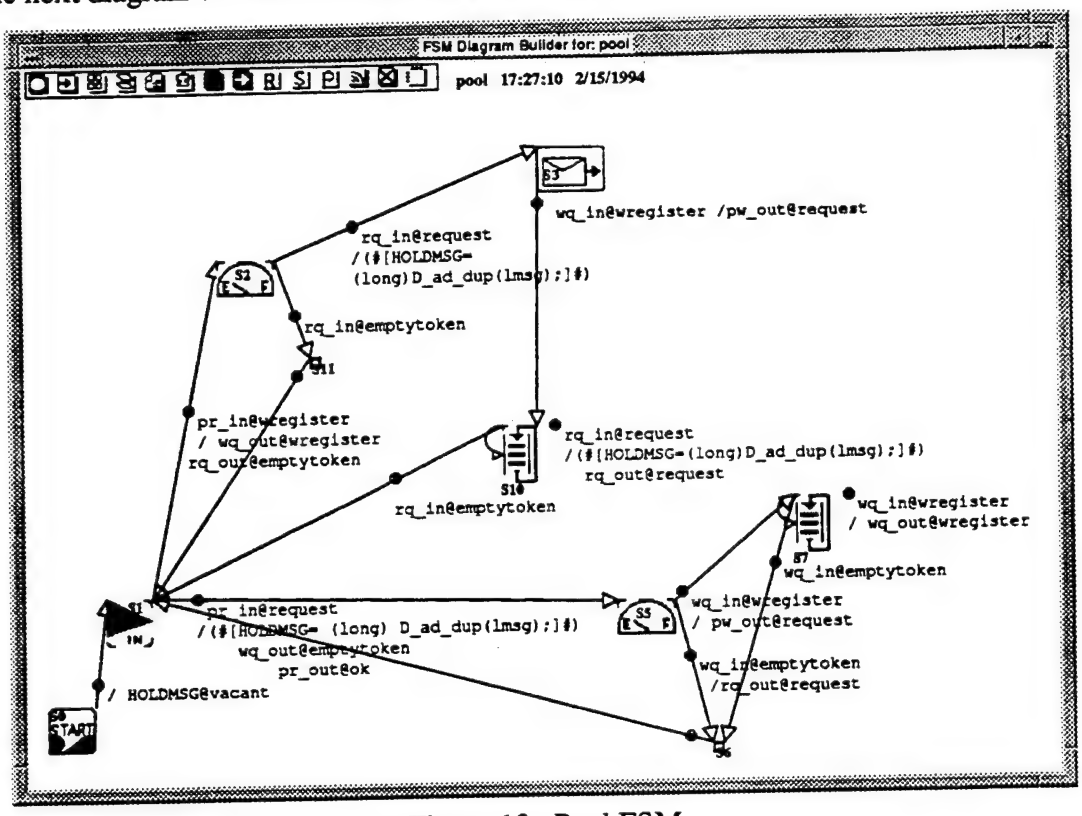


Figure 13. Pool FSM

From the "start" state the pool FSM sets the variable HOLDMSG to the value "vacant". It then moves to state S1, . S1 is the main waiting location. From here the diagram is organized into two branches. The upper branch takes care of worker communication, and the lower branch takes take of requestor communication. In either case the FSM will return to state S1 to process any subsequent messages from the other FSMs.

Upon receiving a "register" message, the FSM puts the message in the wq_out hopper followed by an "emptytoken". It then moves to , S2. At S2 it will check to see if there are any pending requests for the now available worker. If the request hopper is empty it moves to S11 and back to S1. If there is a pending request, it is taken out of the hopper and placed in the variable HOLDMSG.

The FSM then moves to ,S3. In S3 it takes the "wregister" message out of the hopper, using the request message in HOLDMSG, constructs a new "request" message and sends it to the worker. Note that the original "request" message and the "register" message have now been removed from the hoppers.

At this point, the pool manager has performed its function of matching up a worker and a requestor. State S10 is a housekeeping state which shuffles the queue until it finds the "emptytoken" message which is removed. Once this is done the FSM moves back to the waiting-for-input-state S1. Figure 14 shows the one constructor function used to build the pw_out@request message. It takes the message in the HOLDMSG variable and places it in the "msg" variable which causes the message to be send as part of the transition side-effect action.

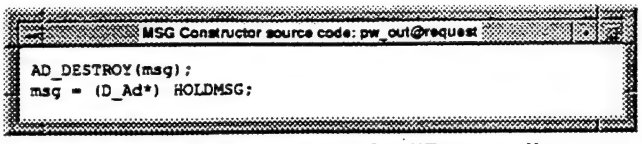


Figure 14. Constructor for "Request"

Lets now take a look at the lower branch for requestors. Upon receiving a "request" the incoming message is placed in a variable, an emptytoken is placed in the worker hopper and an "ok" message is sent back to the requestor FSM. Note that the last message received can be referred to by the reserved variable "lmsg". The FSM then moves to S5.

At S5 two things are possible, a worker is available or it is not. Availability is indicated by the presents of a "register" message in the hopper. The wq_in hopper is checked to see if a worker has registered. If a register message is available then a "request" message is constructed and sent. The same constructor function is used as in the upper branch of the FSM. State S7 is a housekeeping state which shuffles through the hopper and removes the "emptytoken" followed by going back to the wait state, S1.

The constructor is slightly different for wq_out@register. As shown in Figure 15, the last message received is the same message that is put back in the hopper. This can be accomplished by simply putting the "lmsg" into the current message "msg". Note GENERIC-MSG is another name for "lmsg".

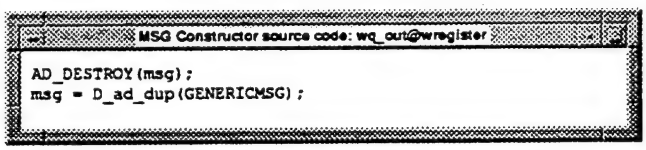


Figure 15. Constructor for Register

If there is no worker registered then the "emptytoken" would be at the top of the hopper. Receiving the "emptytoken" causes the FSM to put the "request" message in the request hopper and returns to the wait state, S1.

For the sake of completeness the message format diagram is included below in Figure 16. This diagram was generated automatically. A default message format was given to each message referenced in the FSM diagrams. As a result of the many-to-one option being chosen for the worker and requestor FSMs, APART automatically includes an identifier field in each message, CID.

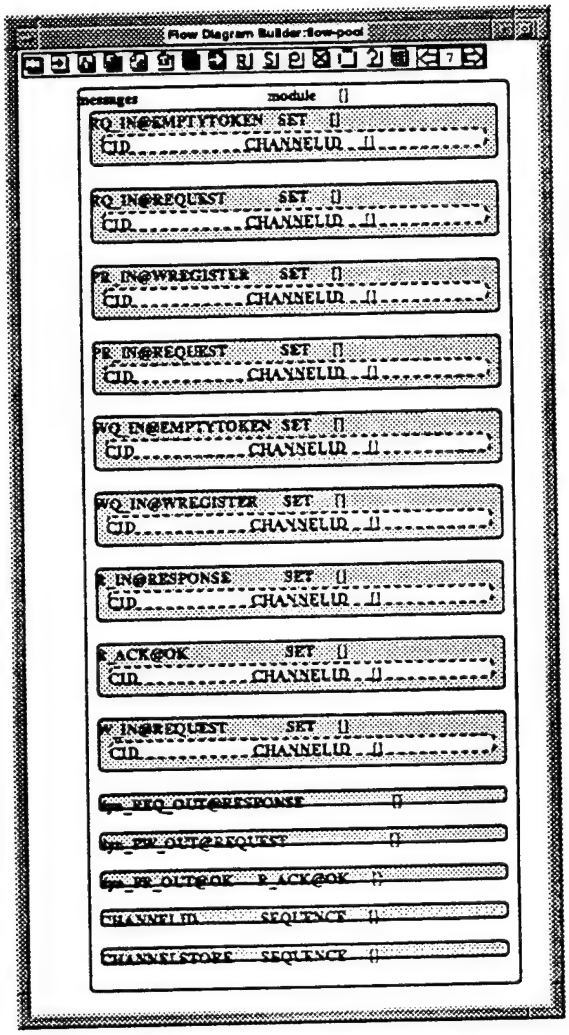


Figure 16. Work Pool Message Format

6.1.6 Reuse Example

The last example in this section demonstrates the idea of reuse. Finite state machine diagrams from different stacks can be recombined and thereby reused as high level building blocks for new protocol. Figure 17 shows this kind of reuse. The flow diagram was constructed by loading both the timer stack and the pool stack which constituted the first two examples. The FSMs are taken out of their respective stack objects and spilled onto the window. The FSMs can then be connected into a new configuration. In this case the timer FSM is connected to the requestor FSM providing a timer service to the requestor. Two additional channels are added to the requestor FSM in support of the new configuration. All the FSMs can then be packaged into a new stack which then contains the super set of protocol behavior.

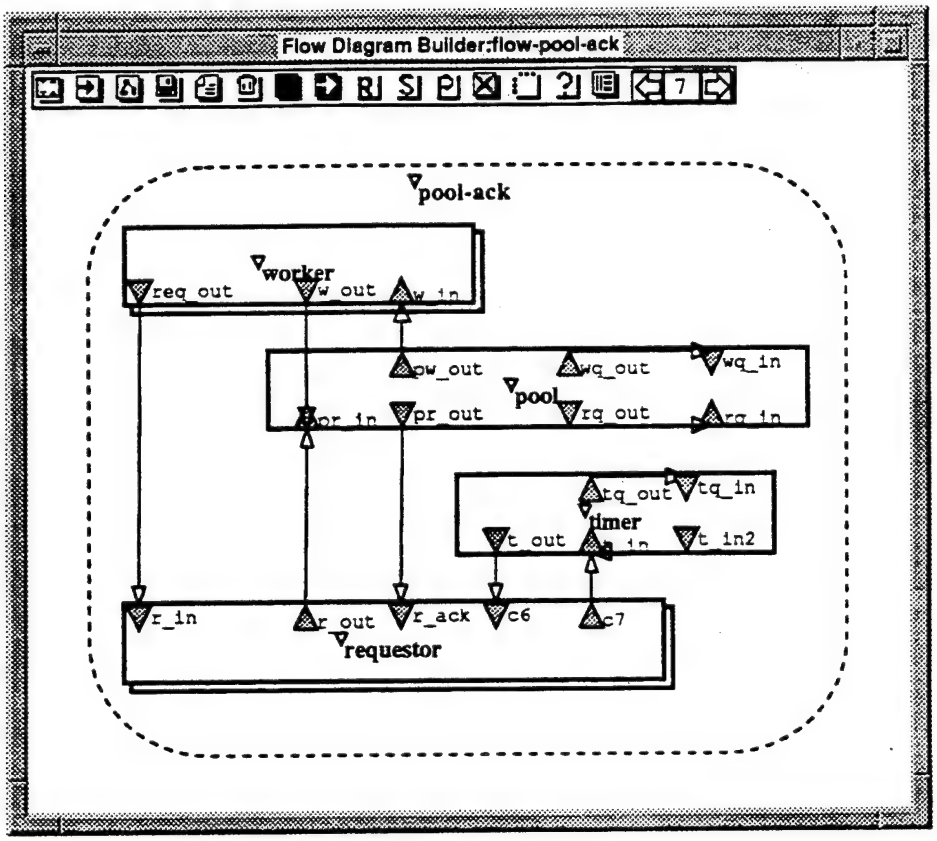


Figure 17. Timer Protocol Reuse

This section was intended to introduce APART and how protocol can be described graphically. Further detail is covered in sections to follow. An enumerated description of all APART functions are documented in the "User Guide".

6.2 Look and Feel

This section will report on how the objectives of look and feel were realized and what was learned in the process.

6.2.1 Basic Building Blocks

As defined in the model section, protocol is defined through use of three interactive graphic diagrams. The diagrams have been named data flow, finite state machine and ASN.1 message format. Each diagram deals with a category of protocol information. Each diagram has its own style and associated look and feel features.

- Finite State Machine

The finite state machine diagram was developed first. The design objective of this diagram was to provide simplicity while being powerful enough to represent all the details of a robust protocol. In this endeavor the team was looking for a solution which provided a clean division between the key functions of protocol and low level detail specific to a particular environment or application. Key functions have to do with state of a protocol. State or memory organizes, controls, and limits protocol behavior. Such control if faulty can lead to deadlock conditions. The team wanted to make sure that this category of behavior is clearly visible in the APART tool environment. Functions which do not have state characteristics fall into the second category. Such functions are much less likely to cause protocol failure. Such functions are assumed or guaranteed to complete without waiting on external stimulus. The behavior of these functions can remain less visible.

The graphical building blocks of the FSM diagram consist of: states, transitions, variables, and side-effects. States, transitions, and variables are all visible to validation. The less visible behavior is packages inside side-effects. The internals of side-effects are by definition not visible to validation.

- States

States were given the look and feel of icons (bitmaps). The icons can be customized to convey high level purpose. There is always a trade-off in screen space for a given icon and the graphic detail that can be represented.

Our experimentation included using icons varying in size from 8 to 48 bits on a side. It was found that sizes of 16 to 32 provided sufficient detail yet kept the diagrams within reasonable size. The bitmaps supported are rectangular, but not necessarily square. This provides some interesting variation. Figure 20 on page 35 shows examples of icons.

- Transitions

Transitions which have the same start and end points have an impact on icon size. The graphic for a transition is an arrow, ⊸→. The arrow is a straight line with the base end attached to the source icon and the arrow head attached to the destination icon. In the case where the source and destination states are the same an elliptical curved arrow is used pointing to the mid point of the icons side ↰. For very small icons such a transition arrow becomes hard to see. This has turned out to be the only major limiting factor for icon size. To clearly show return paths in a diagram, dummy states □ are sometimes

added. Dummy states are states with transitions that have no precondition or side-effect. It has been discovered that clarity of a diagram is improved by making such dummy states very small. Figure 18 shows an FSM diagram which uses both the elliptical arrow and dummy states conventions.

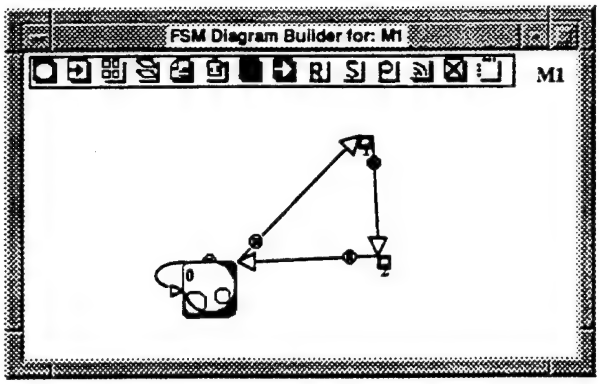


Figure 18. Elliptical transitions & Dummy States

Each transition has an anchor point that looks like a small round button. The button is an active graphic object which can be selected, and to which precondition and side-effect text can be attached. The anchor point and any text attached are all associated with the transition and get dragged along with any movement of the arrow. There can be an arbitrary number of arrows leaving or entering a state. There can be an arbitrary number of arrows between any two states.

In an attempt to keep the diagram as clean as possible, straight lines were used to attach to either the upper left or right of the icon. Graphic constraints are used to dynamically determine which corner to attach the arrow end. Notice in the above diagram the null transitions are attached to the upper right of the icon while the elliptical transition is attached to the upper left.

When transitions are stacked on top of each other, there needs to be a way to select individuals. This capability is achieve through a behavior of the anchor point button. The anchor point button can slide along the transition arrow. When a transition is constructed on top of other transitions the anchor point automatically slides along the arrow so as not to overlap any other anchor points. This makes it possible to uniquely select each transition by choosing the correct anchor point.

Such selection is necessary in support of diagram editing. When a button is selected the transition arrow is automatically moved to the top of the pile of graphic objects. This sets up the condition where you can reliably select a single transition no matter how many are stacked close to or on top of each other. An editing behavior is provided to disconnect and reconnect the end points of the top arrow to any desired state icon. It was found that this combination of graphic behavior is fairly effective while supporting an arbitrarily large number of connections.

- Text Support

One additional behavior was necessary to support the construction of clean diagrams

which has to do with text. In constructing FSM diagrams, one of the major concerns is the layout of text and graphics. For readability, it was necessary, in some cases, to allow the user to move the text and anchor point relative to the arrow. Even when the anchor point is moved off the arrow it is still attached with a relative offset. Any movement of the arrow results in a corresponding move of the text. With the addition of this feature, the goal of being able to construct a readable FSM diagram was achieved.

- Interface to "C" Functions and external packages

Readability is the valuable characteristic in using diagrams. The goal is to provide a complete high level view of a protocol through use of a single diagram. In concert with this goal, preconditions and side-effects tend to be fairly brief and can reasonably fit in the diagram. At the same time it is necessary to support the linkage to "C" source code both large and small as well as entire external software packages and libraries. Such linkage is used to connect the protocol, as embodied in the diagram, to systems functions and external applications.

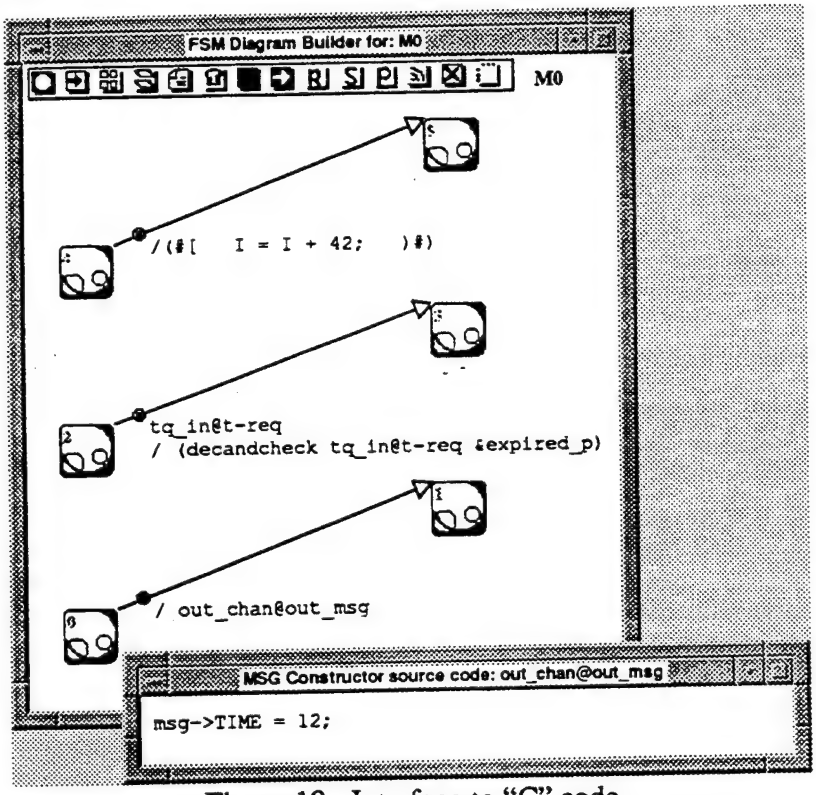


Figure 19. Interface to "C" code

To address the issue of diagram space three ways of attaching to "C" code is provided. If the "C" source code is very small it can be placed directly on the diagram resulting in in-line code. An example is provide in the top transition of Figure 19. This reduces complexity for the very simple case. Another scenario is in message construction. Whenever a message is to be transmitted, the normal behavior is to execute some function which fills

in the fields of the message. This being the typical case, there is specific graphic support for it. By clicking on any output message name in the diagram a text window will be opened. This window is initially empty, but will contain code to fill in the outgoing message. An example can be see in the bottom transition of Figure 19. This code can be thought of as a method in the object oriented paradigm. It only needs to be defined once, but whenever the particular message is transmitted the constructor function will be called. Constructor source code is integral with and saved with the diagram

For very larger source files it is more advantageous to save them separately from the diagram. Function call syntax provides the linkage. These separate files called "stub files" can be edited and recompiled external to the diagram. An example can be seen in the middle transition of Figure 19.

These various ways of combining code provides flexibility of integration. It also makes it possible to combine diagram generated code to traditional text based modules.

- Key Strokes Efficiency

One of the goals in diagram construction was reduction of key strokes. Reduction of key strokes is one measurable way to evaluate ease of use. In pursuit of this goal the team invented the cookie-cutter behavior. The first action is to choose which type of component you want to construct, be it a state or transition. By pressing on the state icon in the main menu, an icon is produced and attached to the cursor. The user can then move the icon to the desired location. Clicking the mouse causes the icon to be deposited at the current location and another icon to be manufactured which again is attached to the cursor. This depositing and duplication gives the appearance similar to stamping out cookies. The cookie-cutter mode is exited at any time by click the middle mouse button. A similar sequence is supported for transition arrows with the added ability of attaching to icons that happen to be under the cursor at the time the mouse is clicked. State icons remain active objects. They can be dragged individually or as a group. Transitions act as rubber bands and stretch between their source and destination. Once activated states can be manufactured and deposited with a single key click. Transitions with two key clicks. It has been our experience that the cookie-cutter behavior makes diagram construction efficient.

To customize icons a palette of icons are provided as can be seen in Figure 20. A drag and drop behavior is used. Clicking on an icon in the palette will cause it to be duplicated and attaches to the cursor. This new icon will replace any existing icon in the FSM diagram when it is dragged and dropped on top. This is a fairly intuitive behavior.

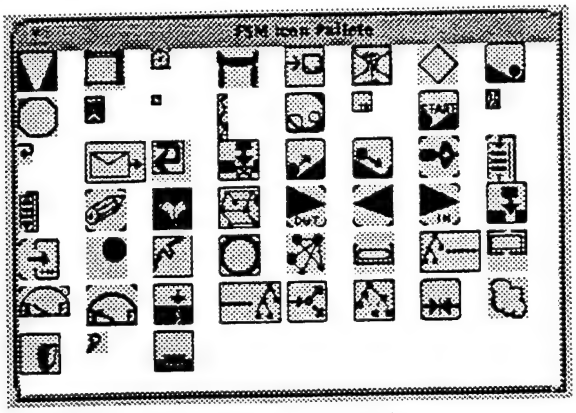


Figure 20. Icon Palette

6.2.2 Overall Meaning of the FSM Diagram

The task at hand is to provide a well structured, meaningful, machine processable definition for protocol. This is sometimes in contrast with protocol that people may devise. People can deal with uncertainty and infinite values. To bridge this gap, our strategy includes the idea that some portion of a protocol definition may remain unbounded, but still provide useful order to the process of protocol construction. This is achieved by dividing up protocol behavior into two categories. The first category is all the behavior explicitly visible in the diagram as states, transitions, and variables (messages being a type of variable). This set of information is visible to validation and well defined. An associated attribute of this category of information is that it is computable within some reasonable time. Information which is unbounded is placed in the second category. The second category consists of environmental variables and variables with very large range of values. This second set of information is carried in side-effects. Selected portions of category two information may be feed back as variable values, but most would remain behind the scenes and therefore not visible to validation. These two categories exist for practical reasons. To achieve predictable protocol behavior it is also necessary that side-effect functions adhere to certain guidelines. It is this adherence that makes the diagram a true representation of the protocol and an accurate high-level description. The next few paragraphs will discuss how to formulate side-effects that adhere to such guidelines.

Diagrams need to support the notion of deterministic behavior. When given a diagram, only one interpretation of behavior should be valid. To fully achieve this goal, it is necessary for the user to restrict himself to common patterns of usage. One such pattern requires that side-effect functions be atomic. That is if the precondition is true then all of the side-effect function is performed. Another requirement of atomicity is that side-effects are completed before the state transition is completed.

Some patterns of usage are more for readability then determinism. One such pattern

requires that a variable not be set and tested within the same transition. This would constituted hidden states. Such states would not be visible to validation. Other patterns are visible to validation, but may be difficult for humans to deal with and therefore should be avoided. An example would be, from any state that all preconditions should be mutually exclusive. If more then one transition could be true at the same time, then it is unpredictable which transition would be taken. The resulting protocol may very well behave properly, but most likely would lead to confusion on the part of the reader.

The clear separation of the two categories of behavior directly impact the practicality of protocol validation. As a goal, the team wanted the visible category to represent all protocol behavior and at the same time wanted to reduce the number of variable values. It is our observation that humans tend to keep a limited number of details in mind at any given time and therefore protocol tends to be limited. This is not always evident from the way protocol has been defined and documented. It is the tool user, designer, responsibility to convert what might appear to be unbounded protocol features into well formed behavior. Notions associated with qualitative reasoning can be applies in this situation. Given a variable with many values, 0 to 2*32, there are usually only a few decision points. This typically manifests itself by checking for boundaries. It is only the boundary values that need be visible for validation. This greatly reduces the number of significant variable values. In the case were there are no obvious boundaries reasonable arbitrary ones can be applied.

6.2.3 Flow Diagram

The flow diagram provides the highest level of abstraction with which a protocol can be viewed at a single glance. The flow diagram contains the organization of the protocol components. Such organization compartmentalized the protocol definition. In object oriented style this provides the possibility of adding, deleting, or reorganizing components without having to reemployment the details.

The flow diagram packages up a group of FSMs, the FSM organization, and an ASN.1 diagram into a composite object called a stack. A stack can be saved on disk as a persistent object. The persistent object contains all relevant protocol information, the graphics as objects and their geometry as well as protocol state behavior, message format, and associated side-effect source code.

The combination of Flow and FSM Diagram provides the user the ability to move from high level to low level. By clicking on a component of the flow diagram a window is opened showing the details of a finite state machine using an FSM diagram. Such graphical behavior has a hypertext flavor in that you can request to see the details of selected portions in any order.

With these few building blocks both simple and powerful application architectures can be constructed. The examples in the Model section are offered as evidence that starting with the most simple distributed application one can progress to client/server and workflow architectures of arbitrarily complex design.

6.2.4 ASN.1 diagram

Message format, the ASN.1 diagram is the third and final type of diagram used to define protocol. The look and feel goal of this diagram is to provide a convenient way to construct and edit arbitrarily complex messages. It was designed to handle the components of the standard notation called abstract syntax notation one, ASN1. Even though there is an easy mapping of one to the other it is not tightly bound. In order to construct messages the user is not required to know the detail encoding rules for ASN.1. The user deals with generic building blocks made out of "sets" and "sequences" of primitive items. Primitive items are represented as rectangles. Primitive items can be grouped. Such groups are referred to as structured items.

Five basic structured types are given unique graphic shape. The five types are: set, sequence, choice, set-of, and sequence-of. Graphic examples of the five types are presented in Figure 21 below:

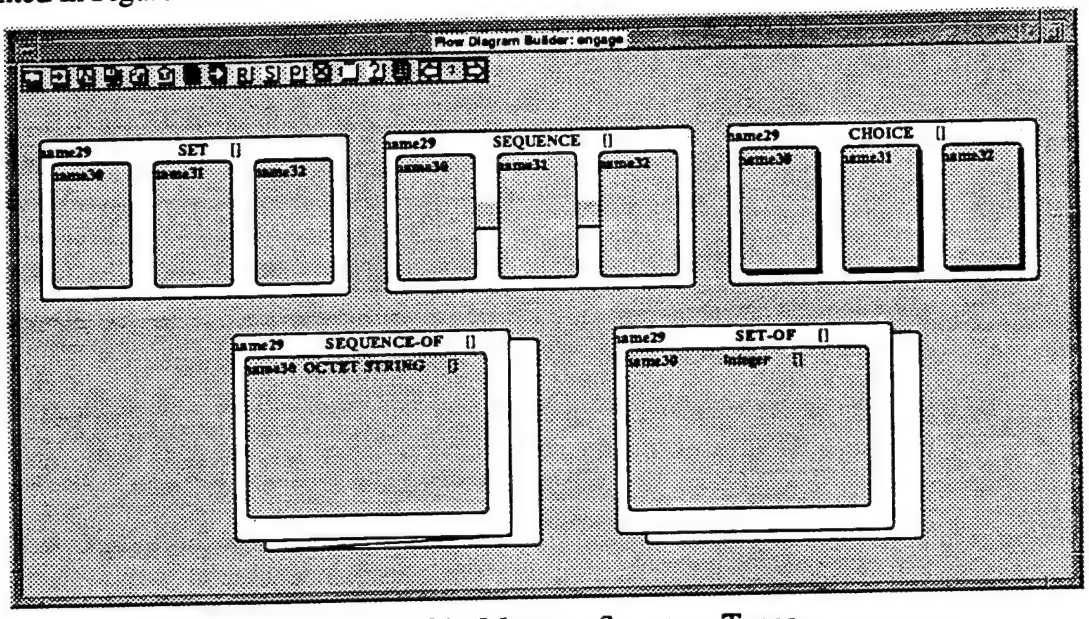


Figure 21. Message Structure Types

One of the prime design issues in displaying complex objects is screen space. The team wanted to make the most advantageous use of screen area which is a very limited resource. One notion was to proportion the objects based on their complexity. A more complex item would be given more screen area. In Figure 22, the diagram on the left is set to equal sizing while the diagram on the right is set to proportional sizing. With proportional sizing, there is a better chance at seeing all the detail at a glance.

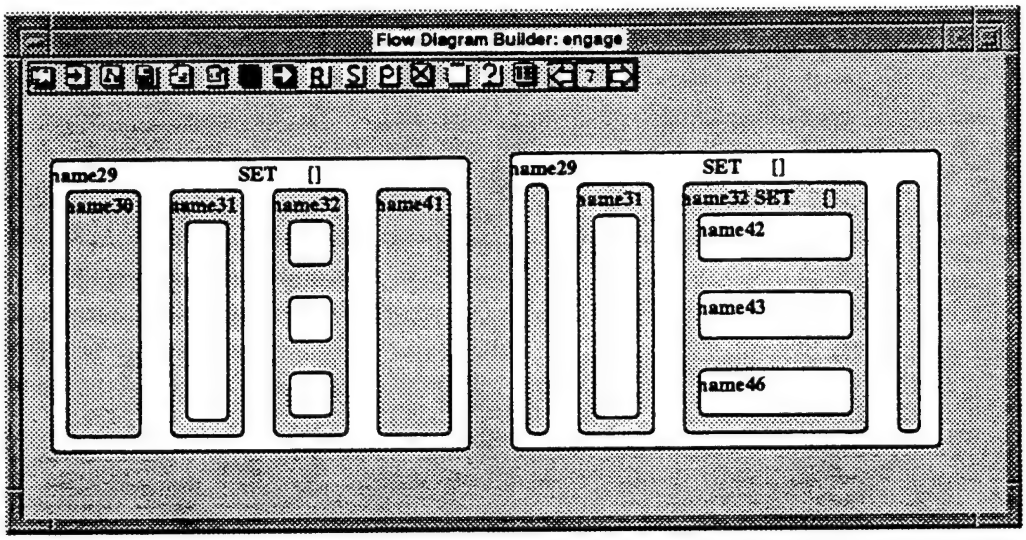


Figure 22. Proportional Spacing Example

In some cases, the complex items may not be the ones of most interest. A notion of shrinking was used to accommodate this need. In Figure 23 below the complex item on the right was marked as shrunken. To determine which objects have been shrunken they are marked with a hatched pattern. The result is that no matter how many sub-items are present the object is sized as if it is a simple primitive.

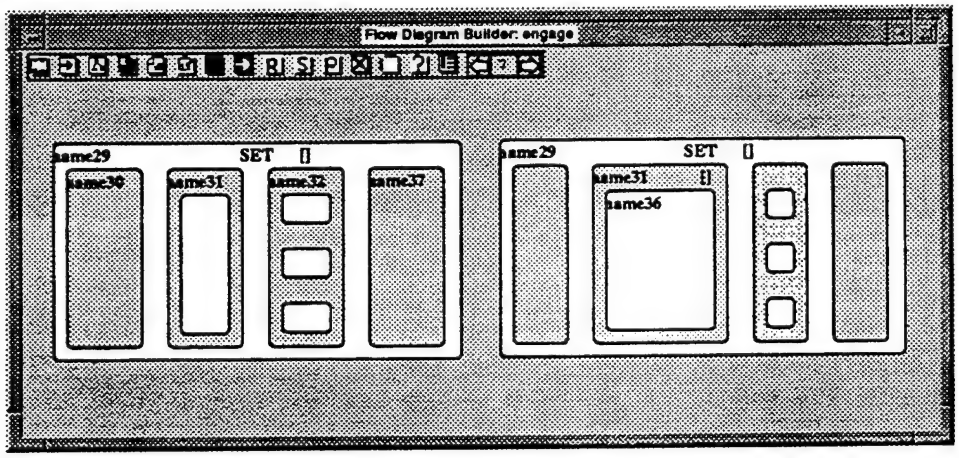


Figure 23. Shrinking of Complex items

Message format diagrams support the user in concentrating on organization without having to constantly deal with item detail. The primitive items are usually integers, character

strings, or a sequence of octets (8 bit packages). During the design process, the user deals with them as logical nested items. The nested structure supports the cutting and pasting at arbitrary levels of abstraction. The ability to grab structured objects as single items reduces the likelihood of error when making adjustments in message layout. In Figure 24 on the left an object made up of three sub-parts has been duplicated with a single key-click. The right shows the results of pasting the duplicated object into object "name6". Notice how the objects have been resized automatically.

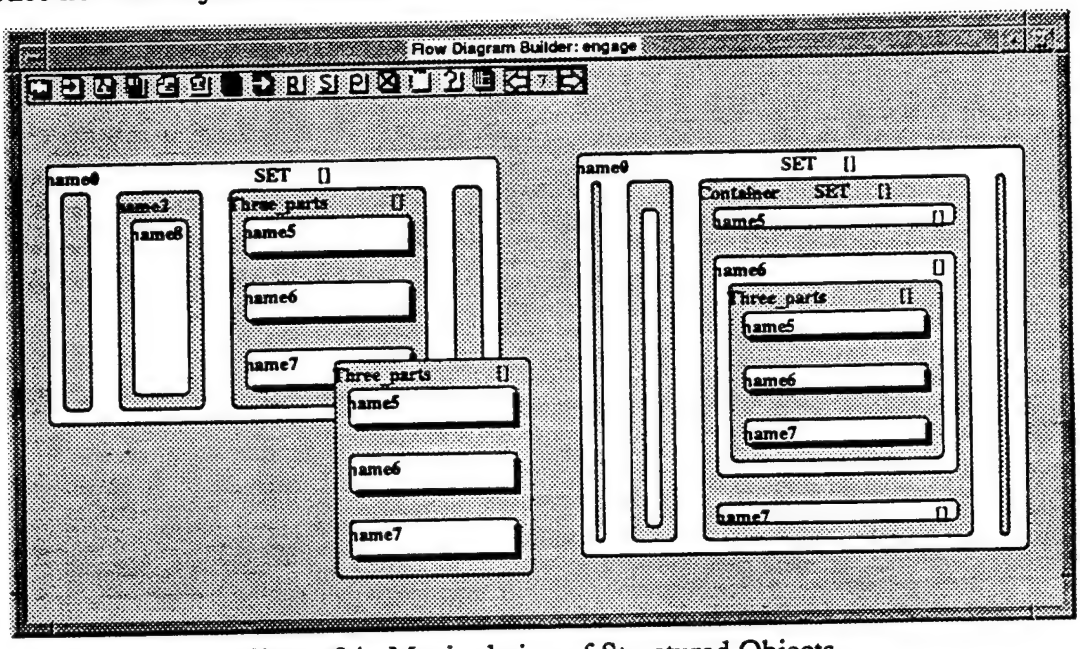


Figure 24. Manipulation of Structured Objects

To conveniently manipulate message format, approximately thirty edit functions are provided. A goal for diagram construction was to gain rapid access to any of the functions while at the same time avoiding the necessity of memorizing many key strokes. This was achieved with the use of three floating menus and a stylized cursor.

The custom cursor, 

, suggests the function contents of each menu: moving, editing, or option setting. One menu is attached to each of the three mouse buttons. Any of the thirty functions can be executed by one mouse down, drag and release sequence. So long as the pop down menus can be activated rapidly even experienced users can stick with the menu interface. If menu activation is quick enough, the user treats the pop down menu as a hand action without having to read the individual items. It becomes ingrained that a left-mouse-click and third item performs a desired function. Figure 25 below show the three pop down menus.

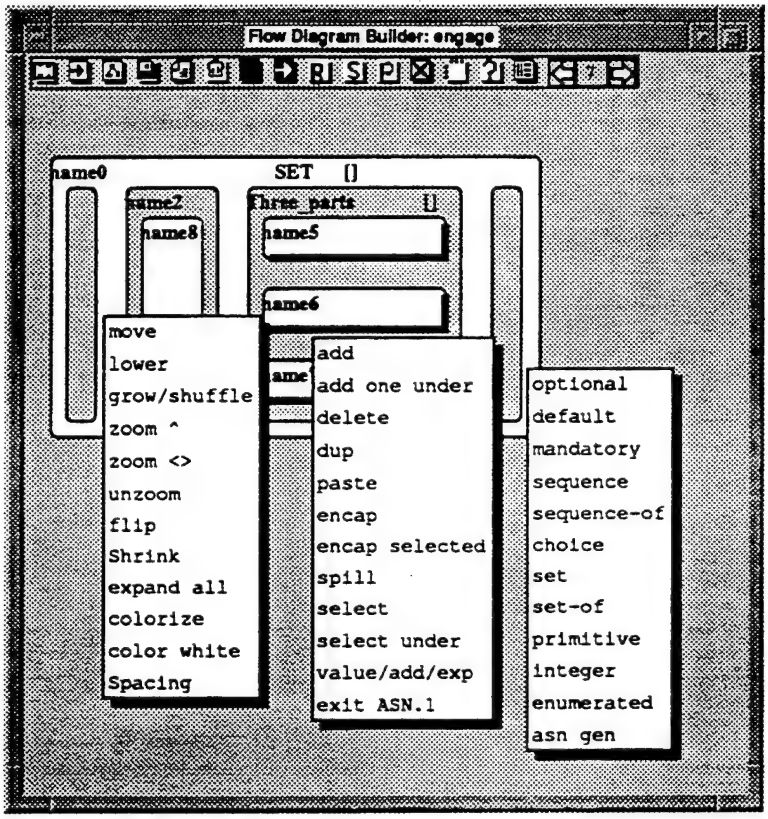


Figure 25. Pop Down Floating Menus

6.2.5 Ease and Speed

The productivity of graphic diagrams went a long way towards the goal of speed. There are fare fewer keystrokes needed to construct a finite state machine by way of diagrams as compared to if-then-else textual statements. The diagrams are a higher level of abstraction with fewer possibilities of error during data entry. Information in the three diagrams can be cross checked.

A particular productivity enhancement was gained when it was found that some missing information could be automatically added. At the time of information gathering, in preparation for code generation, if messages were referenced but not defined, default format was added to the diagram. This provided graphical feedback to the user and also saved data entry time. Consistency between finite state machine diagrams and the set of messages were assured. During initial passes of protocol design, such default messages make early exercise of protocol possible with a low investment of labor.

A factor for ease of use is readability. The speed with which you can enhance or modify a given protocol is tied to how fast you can learn how the implementation is organized and where particular functions take place. The diagrams provide the big picture for the overall organization. The state icons provide high level clues as to where functions are located. Detail implementation can be directly accessed by way of hyper-text functionality. This reduces search time when trying to find the associated low level functions. The aggregate of such functions all aid in readability.

Reuse of protocol information provides another factor in support of speed. One example pertains to message construction. The same message may be sent from different states. Each time a message is transmitted it needs to be constructed. Construction is the filling in of message fields with specific values. It is likely that for a given message type, format, the construction will be mostly the same. The FSM diagram supports the notion of message constructor. Similar to a method in object oriented programming, the constructor need only be defined once for a given message type. The message may be sent from different states, but the same method will be applies unless over written.

6.2.6 Chained Mouse/Graphical Behavior

Interactive behavior can be initiated in two ways. The first it through a mouse action. This is typical. A second way is through a function call which simulates a mouse action. The same interactive behavior would be elicited in either case.

The cookie-cutter behavior of the Tool was made possible by this alternate mechanism. To place multiple icons in a FSM diagram rapidly, the first icon placement interaction is elicited by clicking on a button. This starts a behavior which constructs an icon object and attaches it to the cursor. The icon object will follow the cursor as it moves in the window until the next mouse click. This second mouse click triggers the depositing of the icon at the current position. In addition it manufactures an event which looks to the system like the original mouse action. This causes the sequence to start once again. This sequence can be repeated indefinitely until a particular mouse action signals termination.

6.2.7 Graphic Objects Represent Protocol

The Tool paradigm provides graphical objects to represent protocol. The graphical objects are made up of slots which hold protocol information and graphical behavior. The graphical behavior primarily deals with supporting the user interface during the construction and editing process. Based on the defined meaning of the graphic objects, the code generator performs a translation to executable source code.

A given translation depends on the target environment. The target for the first code gener-

ator was the Distributed Systems Generator[1]. For this package a set of files were generated which conformed to the defined application programming interface, API. This particular API presented an application level service based on the Remote Operation Service Element, an OSI standard. At the application level, message format is typically in ASN.1 notation using ASN.1 basic encoding rules. The same set of objects could be translated into code for other environments.

6.2.8 Persistent Objects Store Protocol

A major effort in the APART tool design was the ability to store protocol components on disk. In this implementation of APART, protocol components take the form of Garnet graphical objects. Such objects contain a complex nested structure of parts. Each part can be an object itself. Objects are made up of mouse-key definitions and behavior functions, location and geometry, associated text of each object, associated bitmaps of some objects, and object relationships. In addition to the various types of information that needs to be saved, the team wanted to be able to handle multiple instances of a given saved object. This raises the question of how objects are named and uniquely identified. The team also wanted to be able to combined saved objects into new objects. This implied that objects may have to be re-named to support the re-packaging function. The names and addresses being described here are for internal use only. The user is not directly aware of these names. These are not the names the user gives to the FSMs and channels.

Each object present in APART has an internal name and address. A protocol stack is made up of many objects. There is no guarantee that names will be unique between one session and the next or between two instances of APART. To avoid name conflicts, the APART tool combines all the protocol objects for a stack into a single complex aggregate object. Once this is done all objects of a given stack are then members of the same object hierarchy.

During the combination process, each object obtains a unique relative name, address, with which they can reference each other. Such references are used to record the object organization. That is, which state transitions point to which state icons and which channel connections point to which channel objects etc. The fact that these are relative addresses means that more then one instance of the same stack can be loaded into APART without the relationships getting tangled up. At APART execution time when stacks are loaded the objects are once again unbundled and given unique name and addresses relative to the execution environment.

The unbundling is necessary for several reasons. The first reason is that objects get distributed across several windows depending on which objects they are, and a Garnet object can only exist in one window at time. For example, each FSM diagram of a stack is placed into a different window. Another reason is due to object clipping. If one object is contained in another it visibility is limited to the boundaries of the parent. In order for the ASN.1 diagram to appear outside the stack object, it needs to be unbundled. In some cases, it is just more efficient to use absolute links between objects at run time. The final

1. A product of the Distributed Software Engineering Tools (DSET) Corp.

reason is in support for the reuse of FSM diagrams. The Tool provides a spill function in which FSM objects of a stack are taken out of the stack object and permitted to exist on there own in the window. In this way, FSM objects can be gathered up and included in other stack objects.

Bitmaps are not actually saved into the stack object when put on disk. Instead the file name of the bitmap is saved in the object slot. When the stack is re-loaded, the bitmaps are read from the standard directory and thereby only one copy of each bitmap is ever stored on disk.

The above mentioned design results in the ability to store, retrieve, and reuse persistent objects in protocol design. The value of this functionality becomes more obvious when a team of people in parallel each build a portion of a protocol stack. During the integration phase FSM diagrams made at different times with different copies of the APART tool will need to be loaded together. APART is capable of performing this integration task even if the protocol designers, working separately, chose the same names for FSMs or channels.

6.2.9 Overlapping Complex Graphical Objects

As the COS team wrestled with the problem of how to represent protocol graphically the idea of nested objects came up. The team wanted to be able to manipulate the organization of objects by simply dragging one object within another. Further the team wanted to be able to reorganize the objects by simply stretching one object to encompass another.

When two objects are partially or completely overlapped, the question arises which one should be selected when the pointing device is in the overlapping region. Typically this is resolved by supporting the notion of one object being over or under another object. The object on top would be the one chosen. In this case, the team wanted to dynamically support which object is on top without the user having to worry about it.

To support this notion graphically, the moving behavior was augmented to include testing which objects were completely surrounded by the currently active, just moved, object. The algorithm re-arranged all contained objects to be on top of the containing object. This resulted in the ability to select and drag any of the objects no matter how they are nested, arranged or re-arranged in the window. This behavior is part of the data flow diagram. An example of its use is depicted in Figure 26 on page 44. On the left FSM A and B are contained in C. The behavior places A and B on top of C. If the objects were to be rearranged as depicted on the right then C is made to be on top of B while B and C are on top of A.

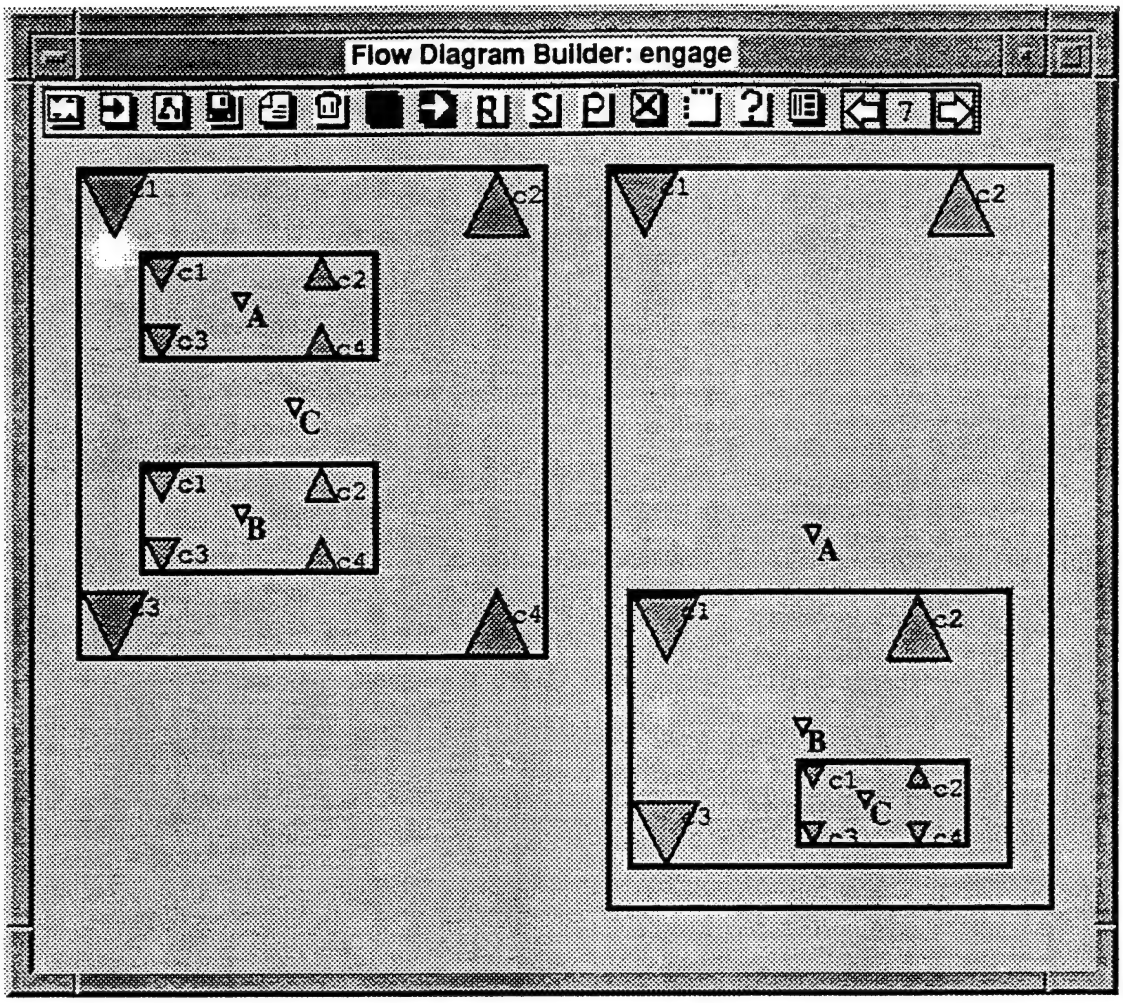


Figure 26. Overlapping Objects

6.2.10 Custom in-but-not-on Behavior

Objects in APART contain other objects sometimes in a nested fashion. For example FSMs are represented as rectangles which contain channels, triangles. As an interactive interface, APART needs to provide a way to select a channel and to select the FSM. Selection is typically done on moving the cursor within an object and clicking. If the user points to a channel by definition the pointer is also within the FSM that contains it. A way needs to be provided to select child objects and parent objects. To accomplish this task a selection behavior of "in-but-not-on" exists in Garnet. "In-but-not-on" means that the cursor is within the objects area and not on top of any child of the object.

While prototyping the interaction between the user and complex graphical objects, a need arouse to be able to select various sub-components based on type. An example can be seen with the stack object. A stack object is made up of several different types of sub-components. There can be a variable number of sub-components each of which can be a complex object as well. In this case, a stack can have a variable number of FSM objects.

An FSM object in turn contains several different types of sub-component objects (channels, title, button, rectangle). It is desirable to be able to elicit different behavior based on which component or sub-component is selected. To accomplish this task, a generic "start where" filter was invented. The filter is a custom version of "in-but-not-on" that works with three criteria. The first criterion specifies which sub-component by type is desired. An example would be to allow the selection of any component of a stack object which is an FSM object. The second criterion accomplishes the "in-but-not-on" function. In this case it only selects an FSM object as long as the cursor is not pointing to a sub-component of the FSM. Such a sub-component could be a channel represented by the triangle. The final criterion allows for the exclusion of sub-components as it pertains to the second criteria, "in-but-not-on". An example would be to exclude the rectangle that represents the FSM. Since the rectangle covers the entire area allocated to the FSM and being a subcomponent of the FSM object there would be no way to be in the FSM area and not on a subcomponent. By making the rectangle an exception to the second filter the FSM object itself can be selected. A filter can then be constructed which selects any FSM object as long as the pointer is not over any sub-component excluding the rectangle. This custom filter provides a convenient way to attach desired behavior to different components of complex graphical objects.

6.2.11 Macro Expansion for Practical Validation

One of the original objectives of APART was to be able to make multiple use of protocol information. In addition to code generation, the team wanted to demonstrate other uses such as validation analysis. As eluded to earlier in the discussion of well formed protocol, additional guidelines may be necessary in the pursuit of good protocol design. This is also true for validation. Before validation can be performed in a real world situation all variables must have computable limits that are within the range of available computer power. It is not to unusual for protocol to have variables with very large range. Even though it mot practical to compute the interaction for all possible values in these situations it is still valuable to do so for a representative sample. To go along with this notion the team wanted to make the designation of the sample convenient. The solution took the form of a macro. Macro in this case refers to a statement in the APART language that expands to one or more transitions. An example of such a macro is covered in the following paragraph.

From a validation point of view, APART considers all variables to be enumerated types. To analyze a variable means to check all state transitions for each defined value. The task at hand comes down to defining all the values of interest. Defining in this case means to add a transition in the FSM diagram for each of the defined values. Such transitions could be added just like any other transitions, one at a time. To make this process convenient, a macro was designed that supports the creation of groups of transitions with a single statement. For example, the following macro statement generates a set of six transitions each with a precondition for a value of variable "C" and with a final transition that sets variable "iszero".

```
(val :var c :range (genseq 1 6) :result iszero)
```

In Figure 27 below the VAL macro is show in its compressed form. In Figure 28 the expanded transitions can be seen. Under normal circumstances the user would not need to view the expanded form.

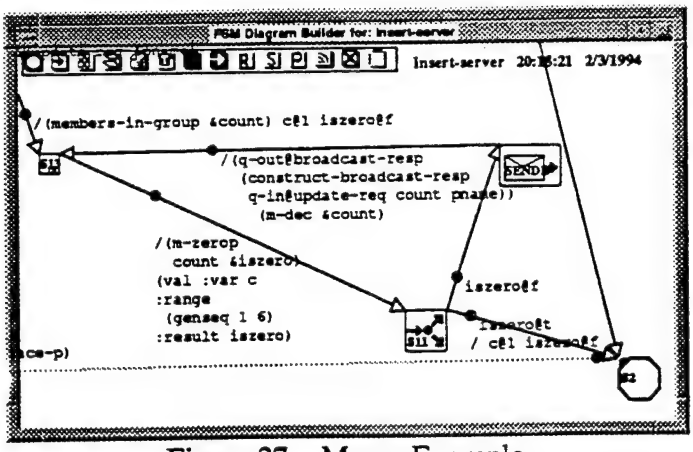


Figure 27. Macro Example

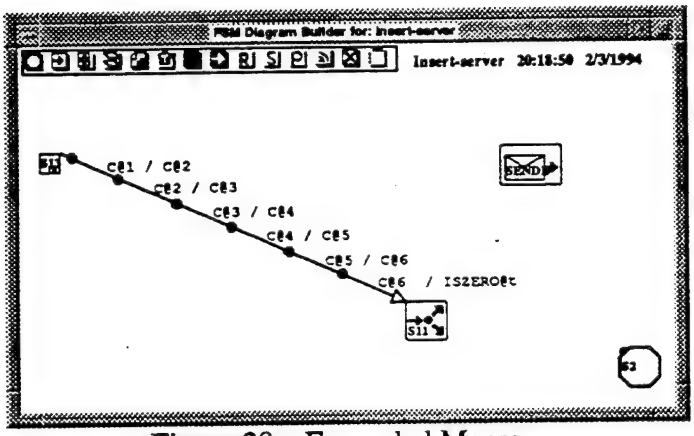


Figure 28. Expanded Macro

In addition to generating the transitions, the macro identifies the transitions to be for the purpose of validation. It is then convenient for the user with aid of APART to include or exclude these transitions. Such transitions would only be included during a validation session and excluded for code generation.

6.2.12 Isometric Projection

One of the objectives of our research was to demonstrate the reuse of protocol information as well as to present the information in graphical form. One of the results of this endeavor was isometric projection of finite state machine diagrams. The key benefit of isometric project is to better present state variables and their role in protocol behavior.

From a theoretical point of view, states and variables are interchangeable, but at the same time they have significantly different visual appearance. Unlike states which have a single

geometric location, variables are global. Setting a variable potentially can effect many points of the finite state machine. This makes it hard to visualize how a variable effects protocol behavior.

Isometric project is a way to give graphical place to variables and their values. Variables can then take on graphical characteristics of states. The idea is based on the fact that each variable value effectively multiplies the number of states. The following windows provide an example of the interplay of variables and state. Figure 29 shows an FSM diagram with four states and one variable. The variable, VAR, takes on three values (42,88,99). From a cursory look at the diagram it seems that there may be an infinite loop by traveling along the path S3, S1, and S2. In actuality the loop is traversed only once ending in a deadlock condition at state S3.

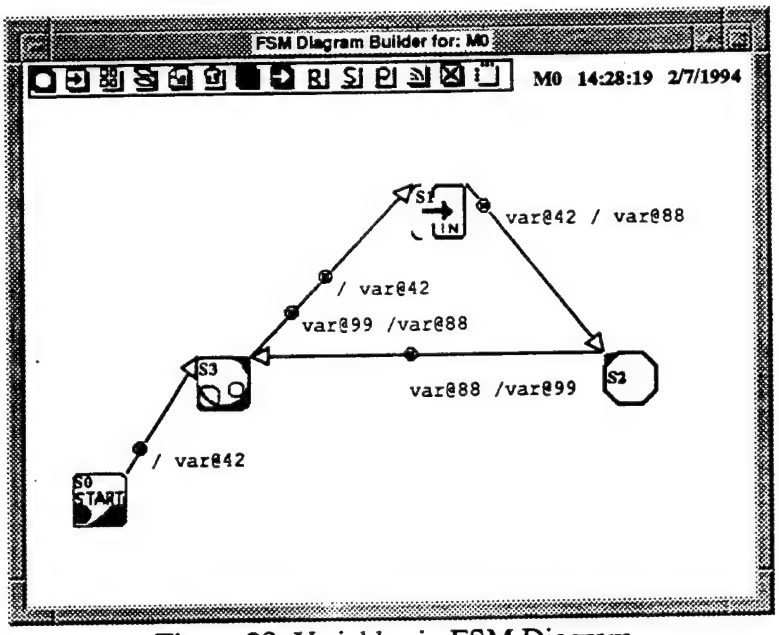


Figure 29. Variables in FSM Diagram

In the isometric projection of Figure 30 three planes are drawn, one for each variable value. A state is drawn on a plane if the state can be visited at the same time the variable is set to the value designated in the plane label. Potentially the original four states can be duplicated three times and shown once on each plane. The state will be shown only if there is a defined combination of state and variable value.

In Figure 30, state S0 only appears on the VAR@42 plane because this is the only combination possible. States S3 and S1 appear twice displaying the fact that the variable takes on two values while in these states. Its interesting to see that state S1 on plane VAR@88 is a dead end. It is not the stop state and there are no arrows leading out. This is the graphic representation of a deadlock condition. The deadlock condition is clearly visualized in the isometric projection. Such diagrams are automatically generated from the original information. This is a clear example of how information can be used in different ways for better understanding.

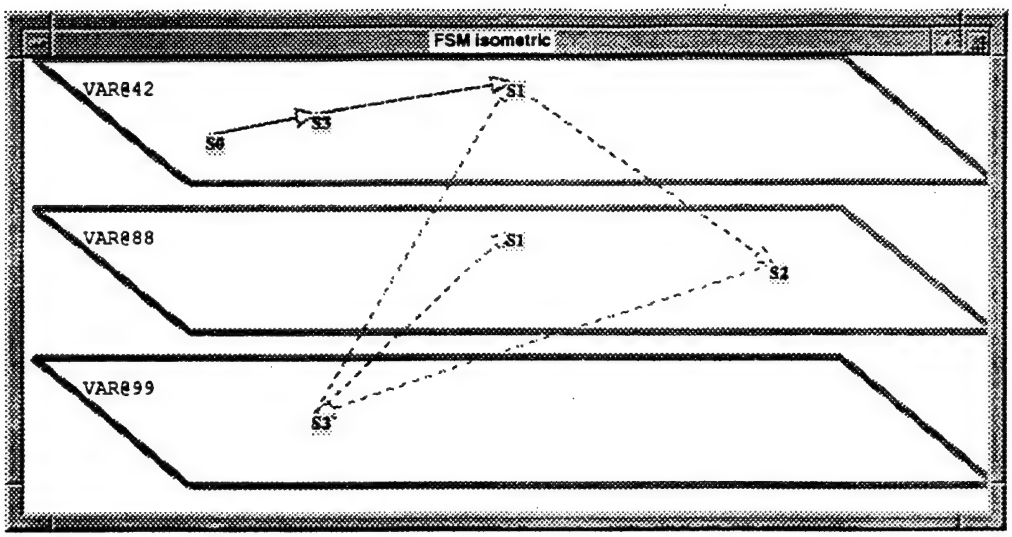


Figure 30. Isometric Projection

6.2.13 Dynamic Menus by Introspection

To provide ease of use, particularly with minimum training, it is advantageous to provide menus in support of the graphic interaction. Being that this was a prototyping project with iterative development, the team wanted to experiment with ways to automatically keep menus up to-date as the graphical objects evolved. If each object had sufficient information stored within it, it should be possible to construct the interface menu by extracting the necessary information from the object programmatically. The added value to this approach is that when the objects are enhanced during the iterative process, the interface menus are automatically updated to reflect the latest set of functions and behavior. A corollary objective assumes that such menus can be constructed fast enough at run time.

The menus served two purposes. The first was to document what functions were possible for a given graphical object along with what mouse key-click triggered the behavior. The second was to be able to trigger the behavior using the menu and thereby not having to directly know or use the assigned mouse key combination.

To accomplish these two purposes, the menus had to contain three pieces of information: a name for the behavior that the user can identify, the key-mouse combination to start the behavior, and the location on the screen where the key-click should be located to identify the correct object. To request a menu, the user points and clicks on the object of interest. The first step in the process is to identify an object at the location of the pointer. The object is interrogated for all valid behavior. For each behavior an entry in the menu is constructed that contains the name of the behavior, the key-click that can activate the behavior along with the original location. The menu is then displayed for the user to make a selection. To make a selection the user has to move the pointing device down the menu. By choosing the menu item the pointer may no longer be over the desired object. To take care

of this condition, the next step in the process is to relocate the pointer to the saved location and then generate the necessary mouse key-click event.

The team found that dynamic menus are possible and can save development time particularly if the iterative development process is used. The team made use of dynamic menus in the flow diagrams. This approach placed some design constraints on the objects. One constraint was that the behavior functions had to be packaged as slots in the objects. Behavior functions needed externally meaningful names since they are made visible to the user in the menus. Since the behaviors are triggered by mouse key-clicks, it became necessary to made sure that there were no conflicts as additional behaviors were added.

Currently the menus are completely constructed each time it is called up. The speed is a function of the number of behaviors associated with the object. Given the current hardware it takes one to two seconds for the menus to appear. This is a little slow. An alternative could be to construct the menu the first time and save it in an object slot. This could provide a significant speed up while still supporting the dynamic menu update capability. An example of a dynamic menu can be seen in Figure 31.

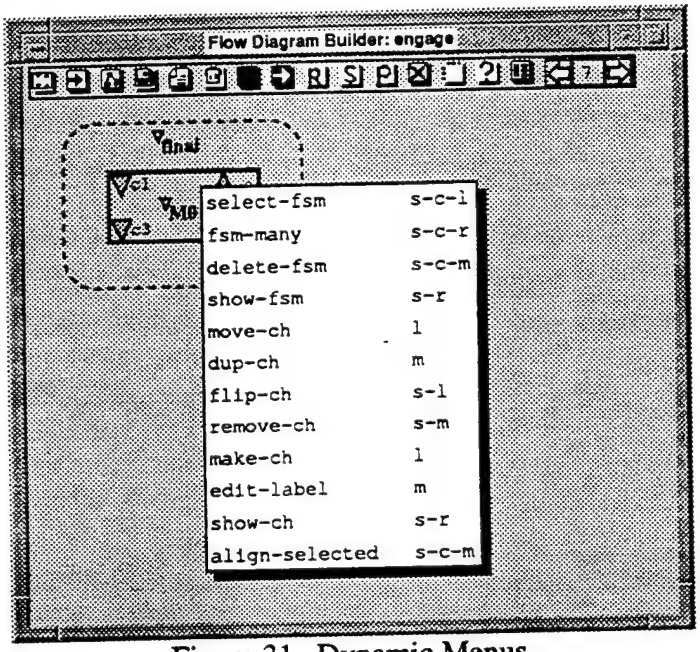


Figure 31. Dynamic Menus

6.2.14 Hyper-text Buttons

Protocol as presented by the Tool is made up of several different parts. The parts are represented as active graphical objects. The team wanted these parts to have an active relationship so the overall organization of the protocol is apparent and supported by the interface. One feature in support of this concept is the tieing of the FSM object in the flow diagram with the corresponding FSM diagram. The user can click on the FSM object in

the flow diagram and cause the FSM diagram to be displayed in its window. If the window did not yet exist it would be created. If the window exists it would be opened and raised to the top of the window pile.

Another feature is the relationship between message names in the FSM diagram and constructor source code. The user can click on the message name in the FSM diagram and the constructor will be displayed in its window. If the window did not yet exist it would be created. If the window exists, it would be opened and raised to the top of the window pile. These relationships help bind the design together and help the user navigate over the various parts of the protocol.

6.2.15 Simulation/Execution

The simulator module provides a means to check out the interaction of all the finite state machines in a stack. This section first describes the look and feel of the simulator to provide some background. Discussion of the design decisions and prototyping experience will follow the basic description.

The Simulator controls the separately executing, distributed, finite state machines through use of an application level protocol. Its unit of control is the state transition. In order for a FSM to proceed from one state to the next, in addition to the defined preconditions, it must also get permission from the simulator by way of the simulator protocol. The simulator itself is implemented as a finite state machine. A protocol has been defined with which the simulator communicates with the FSMs of the stack. As part of the code generation process, if the simulator is present, the code for each FSM is augmented to include the simulator protocol.

The simulator finite state machine is automatically added to the stack when the user selects "simulator load". As part of the loading process, the original FSMs of the stack are augmented with additional channels used to communicate with the simulator. Figure 32 shows how the simulator is connected.

To start a simulator session the "simulator" "run" option is selected. The Tool running as a Unix task starts the simulator FSM as a sub-task. The standard file input/output of the sub-task is routed back to a separate process in the parent, Tool, task. This process waits on data from the sub-task and thereby provides the linkage between the Tool graphics and the simulator.

At this point, each of the FSMs in the stack can be put in execution as a separate running unix task. The first thing these tasks do, as part of the simulator protocol, is to send a message to the simulator. This message consists of the name of the task, a unique object identifier which is used for addressing, the values of any variables defined, and the contents of the last received message if any. Upon arrival, a window is opened to display the status of the FSM which sent the message. In our example stack there are two FSM tasks, Bob and Susan. Figure 33 shows the two status windows for Bob and Susan on the left. The corresponding FSM diagrams are shown on the right. Each FSM is now waiting in their initial state, S0. This is analogous to a break point.

At this stage, the user can press the "GO" button within each status window. This causes a

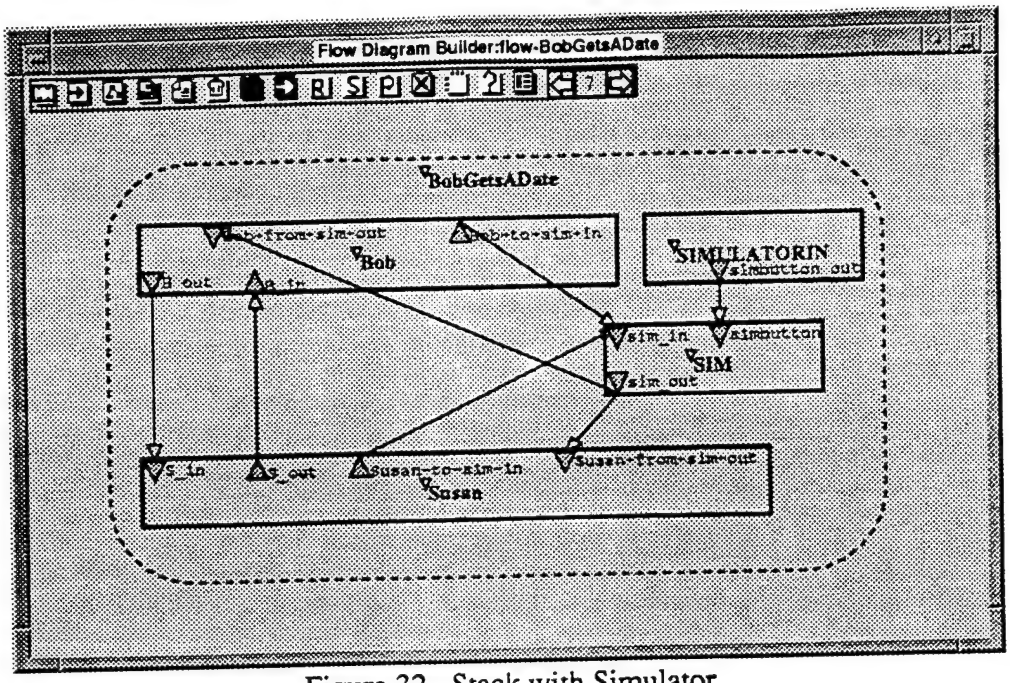


Figure 32. Stack with Simulator

message to be sent to the particular FSM. This action can be thought of as putting the FSM into run mode. By pressing the "GO" button for the Bob FSM, it allows the FSM to send the "WannaGoOut" message to the Susan FSM. At this point the Bob FSM transitions to state S1. State S1 waits on the arrival of either the "OK" or "SORRY" message. The status window for Bob is now shaded indicating that it is in run mode and not waiting on a simulator message. Upon arrival of the "WannaGoOut" message Susan would have transitioned from state S1 to S2. Before Susan can check the "DanceCard" variable the simulator must give the go ahead. This is indicated by the "GO" button in the Susan status window. Figure 34 contains a window group showing the status windows as just described.

Pressing the "GO" button for Susan allows the transition from state S2 to S4 and the sending of the "OK" message. With the arrival of the "OK" message Bob's status window become un-shaded, displays the fact that Bob is now in state S2, and waiting on the simulator to proceed. Figure 35 contains a window group that shows the updated status. This figure along with the previous two provides a glimpse at the simulator look and feel.

The simulator has to support several design objectives. One is the ability to control the execution of the FSMs even when they are widely distributed over different machines and at different locations. The team experimented with the idea that the simulator could be just another FSM in the stack and thereby take on the characteristic of a distributed application. This approach gave the added value that the simulator FSM could be constructed using the Tool itself. In order for the user not to have to deal with the simulator protocol directly, the team took the approach that the code generator would be able to implant the simulator protocol automatically in each user defined FSM. As discussed earlier, a transi-

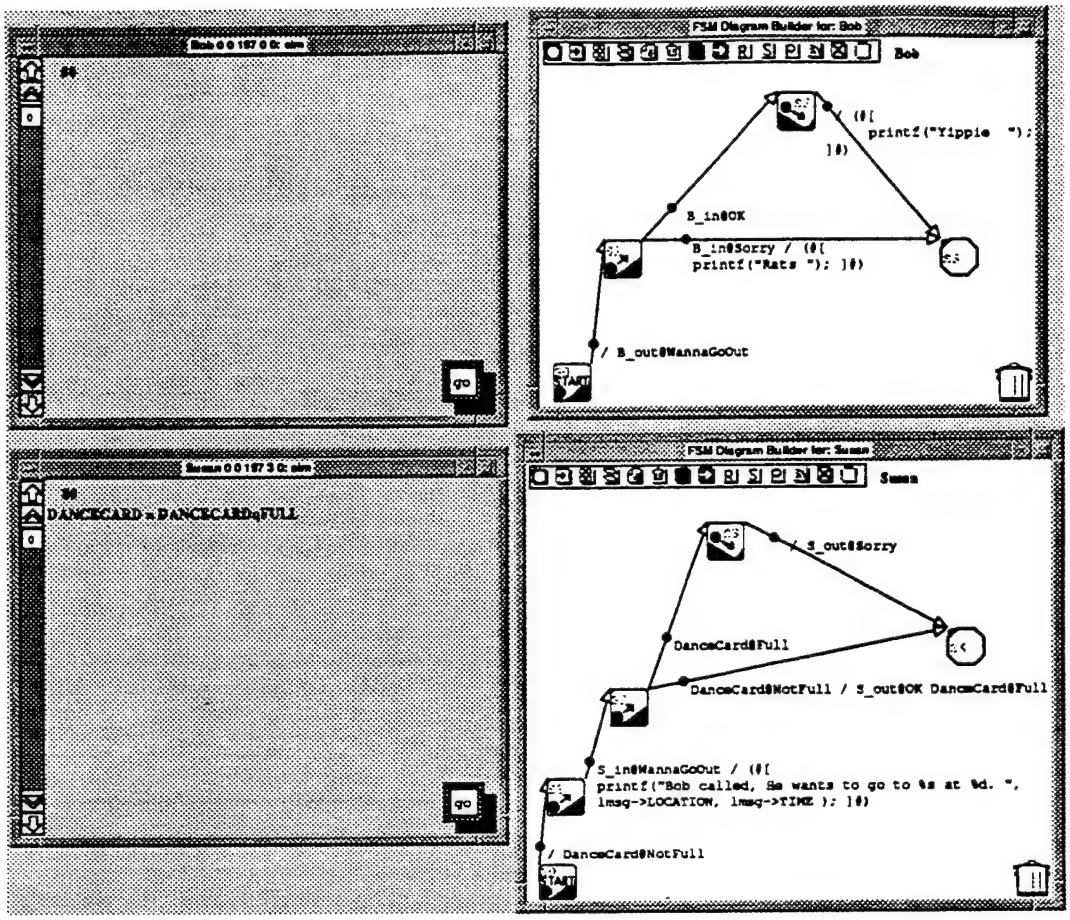


Figure 33. Simulator Look & Feel at Start-up

tion with all of its side-effects are non-divisible. This lead us to define the state transition to be the smallest unit that could be individually controlled by the simulator.

Another major design decision was the graphical look and feel for the simulator interface. Stacks could have a variable number of FSMs. The team wanted to automate the layout and yet support this variability. The team experimented with trying to reuse the information that was already present in the data flow diagram. The user laid out the stack and in doing so organized the FSMs. The team decided to re-used this organization when laying out FSM status windows. This works well in the simple case when there is only one instance of each FSM. In cases where there are many-to-one relationships it was necessary to support the layout of may instances of the same FSM which is drawn only once in the flow diagram. The resulting design trade-off was that the original placement of FSMs in the flow-diagram would be the starting place and the location for the first FSM instance. If additional instances of the same FSM were to appear the corresponding status information window would be stacked horizontally. This works well for a good many cases, but not all. As the complexity of stacks increase this algorithm may need to be revisited.

As indicated by the scroll bar on the left of the status window, it was envisioned to store on

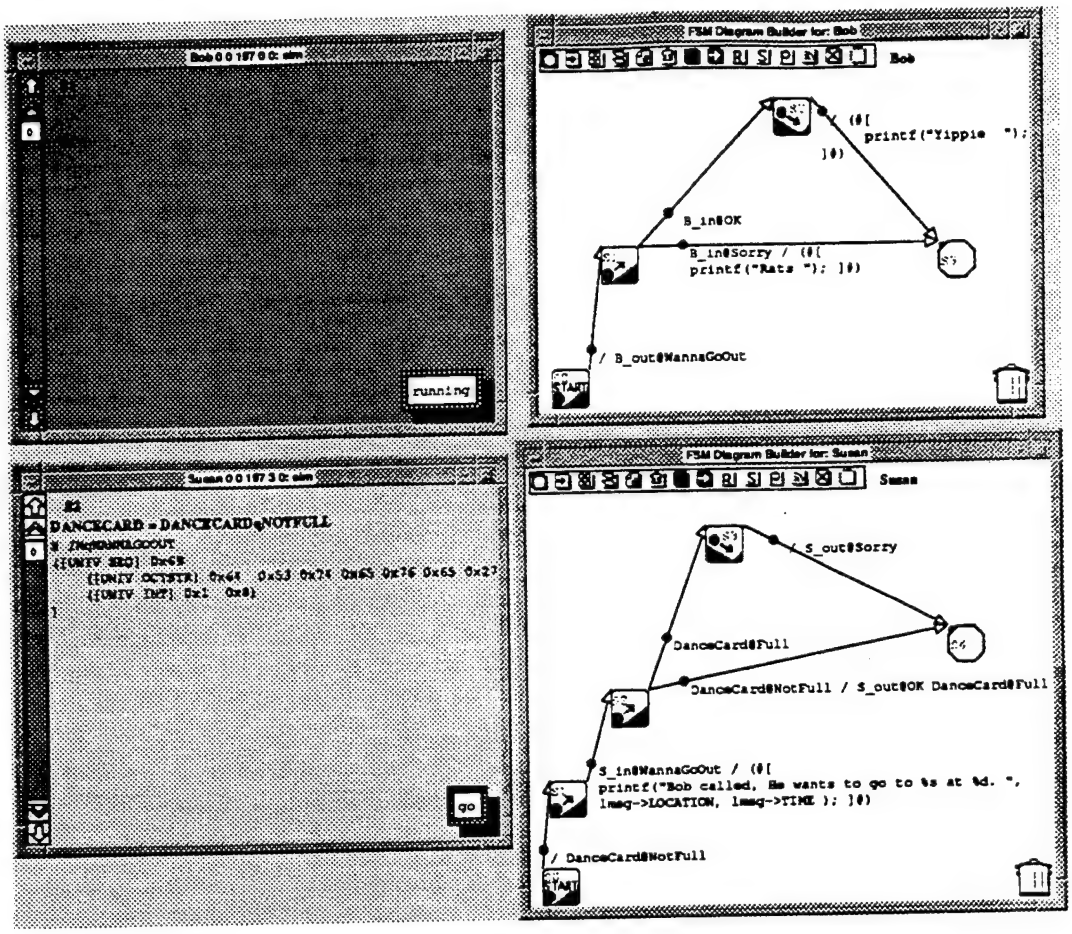


Figure 34. Simulator Window after GO button pressed

disk the incoming status information. This would allow scrolling forward and back of the status information providing a trace history. Time prevented the completion of this function.

While the simulator and FSMs are running, several things can happen concurrently. At the same time that the simulator is running and providing status update, the user can continue to interact with the graphical interface. To support this concurrence the Tool is divided into two light-weight processes. One process continues to service the graphical interface while the other waits on the reception of messages from the simulator task. The two processes have the same address space to share and update the common graphical objects.

Some additional functionality had to be added in order for the FSMs to send copies of its messages to the simulator and on to Tool graphics for display. During the simulator load process, simulator message formats are automatically added to the ASN.1 format diagram. The user does not have to be concerned with message format in support of simulation. These message formats are used by the simulator code in carrying out the simulator protocol. When the simulator task receives a message it decodes it into displayable character

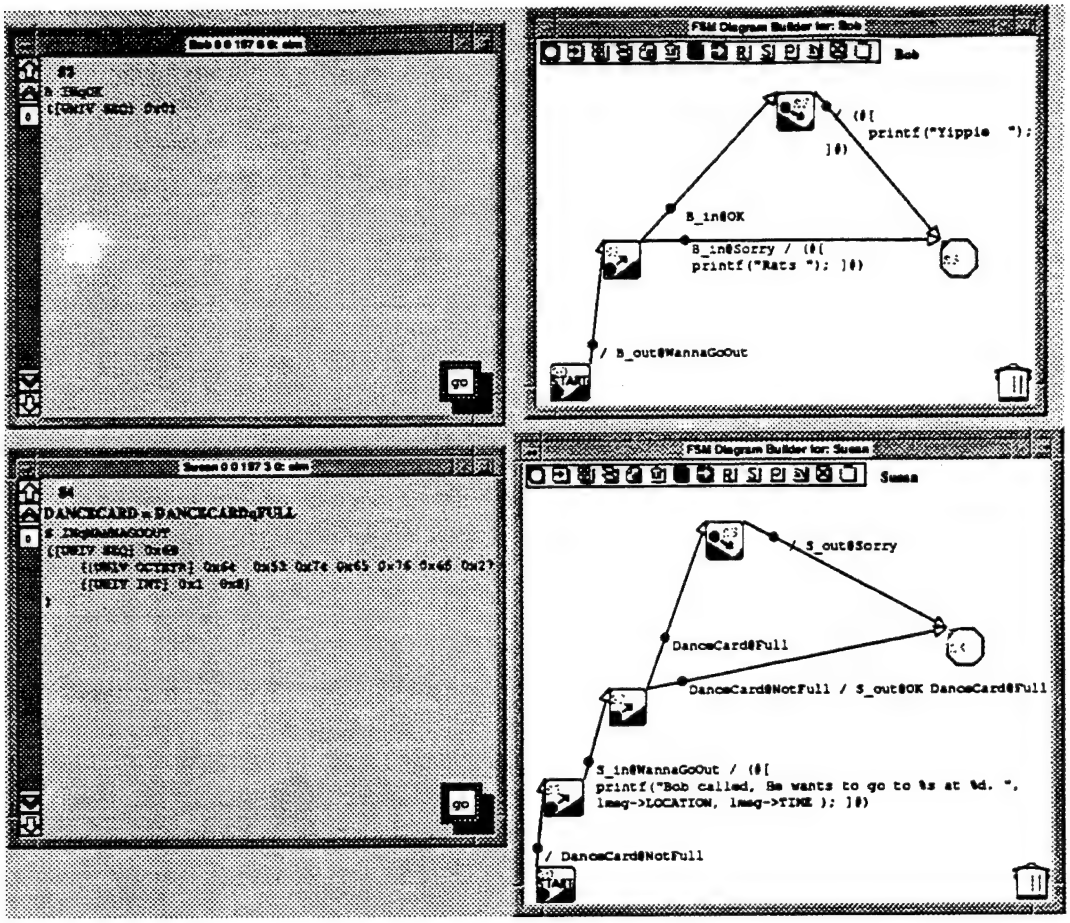


Figure 35. Simulator Status after FSM State Transition

format before sending it on to Tool graphics. This relieves Tool graphics from having to parse ASN.1 messages. Alternate experiments included having the simulator send an ASN.1 formatted character stream to the Tool task, but this proved to be to inefficient.

6.3 Protocol Validation

There are a number of metrics that may be applied to a protocol to validate how well it performs. A few common metrics include: deadlock, livelock, and whether there are unreachable states.

Perhaps the most classic failure of multi-agent interaction is deadlock. Deadlock occurs in communication when both ends of a channel are waiting indefinitely for the other side to do something. Once a conversation is in deadlock, there is no way to terminate. A protocol that can lead to deadlock is poorly designed.

Livelock is less serious than deadlock. Livelock occurs when a protocol allows endless interaction between to agents, without progress. For example, a protocol may allow one side of a conversation to request an abort, and have it denied, request, deny,... etc. indefinitely. A livelock loop must have at least one sequence that would lead to terminate, otherwise the loop would be in deadlock, by definition. As such, Livelock is less alarming than deadlock.

Consider a finite state machine model of a protocol. If there is no path from a start state to state X, what is the point in having X? X is not harmful to the protocol; it is simply unnecessary. More importantly, it may be some unfinished thought of the protocol designer, and so perhaps it should be flagged at least as a warning. Clearly a protocol with unreachable states is less a problem than either livelock or deadlock.

6.3.1 Proof Theoretic Approach to Validation.

To prove some property by hand, it would be necessary to establish that it holds for all execution sequences. An execution sequence is a path (through the finite state machine, following the arcs) that originates at a start state. For example, one may want to prove that there is no execution sequence that leads to deadlock.

In general, Finite State Machines may (and protocol FSMs usually do) have infinite execution sequences. And likewise, it may have an infinite number of infinite execution sequences. And so, proofs are accomplished by induction.

One proves that if, the property holds for an executable sequence of n-1 states, then the property holds for the sequence of n states, where there is a link from the n-1th state to the nth state. Restating this, such a proof establishes that all reachable states from a start state retain the desired property.

Needless to say, manual proofs are rarely used on non-trivial protocols.

6.3.2 Validation Implemented in APART

The primary focus of the protocol validation component of the APART tool is to provide a flexible skeleton for validation techniques. The APART team has implemented deadlock detection on singleton finite state machines as a demonstration of design reuse for computer automated validation.

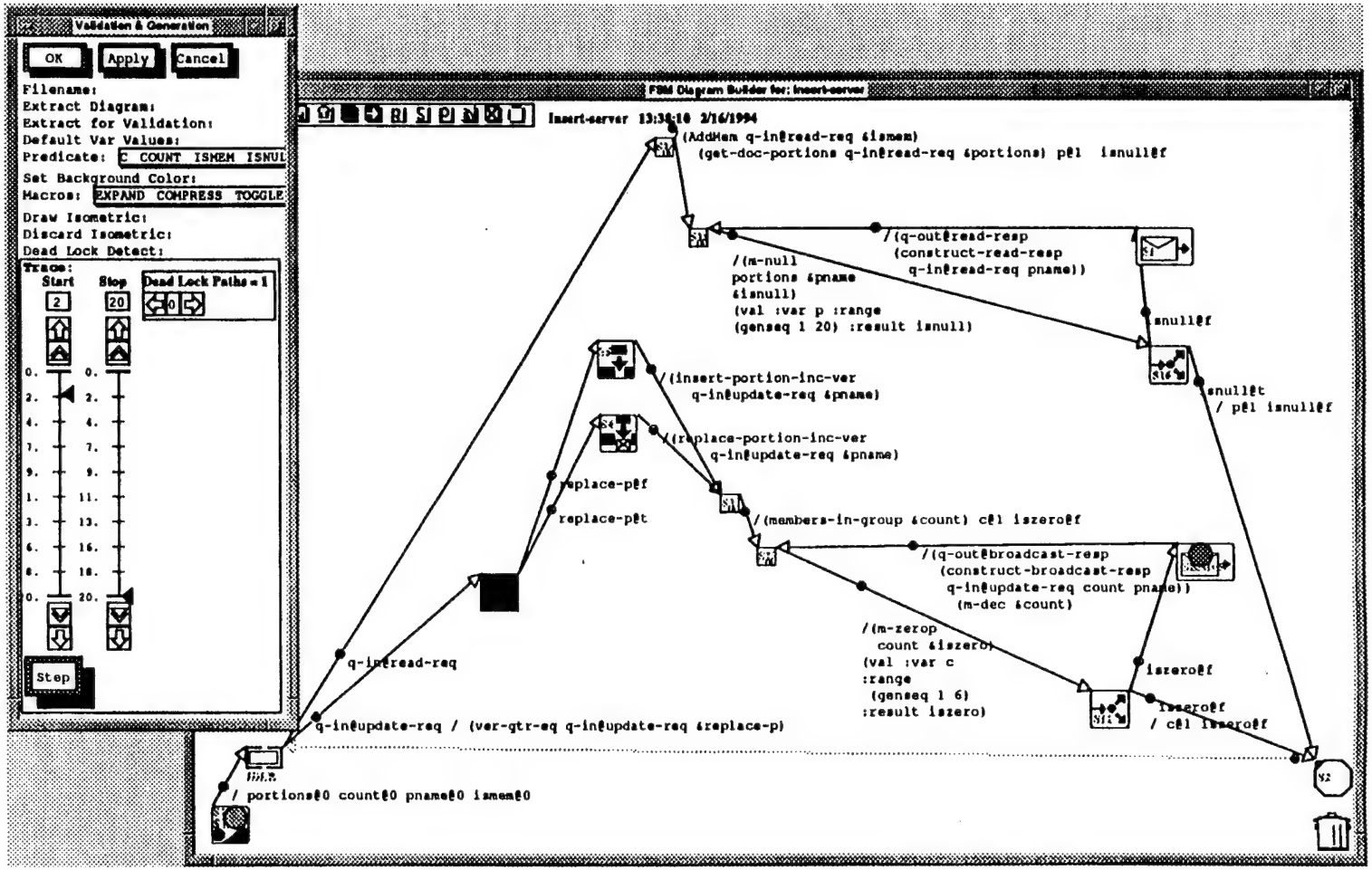


Figure 36. Protocol Validation Graphic Feedback

Following deadlock detection analysis, a validation control panel is displayed as shown in Figure 36. The panel indicates that there is one path to a deadlock condition. It also indicates by the slider bars that there are 20 state transitions in the deadlock path. A green ball indicates the start of the deadlock path which can be seen over the "start state" at the lower left in the diagram. A red ball indicates the "state" where the deadlock is found and can be seen over the "send state" at the right of the diagram. The state in inverse video is the "current" state along the path to deadlock. As can be seen from the left slider, the current state is the second of 20 along the path to deadlock. The user controls the "bouncing ball" to travel along the path from start to deadlock with the slider bars in the control panel.

6.4 Code Generation and Compilation

APART automates the process of going from design to implementation. The final output of the APART tool is a reference implementation of the newly defined distributed systems protocol.

APART translates the collection of diagrams into "C" code. The user then compiles the code into a set of executable tasks with a single executable task produced for each finite state machine diagram. APART provides hooks into the generated "C" code as well as "stubs" files where the developer puts the "C"-functions. So the developer can call any C function to generate or retrieve information from databases, the operating system, or ASN.1 messages. The generated process is depicted in Figure 37.

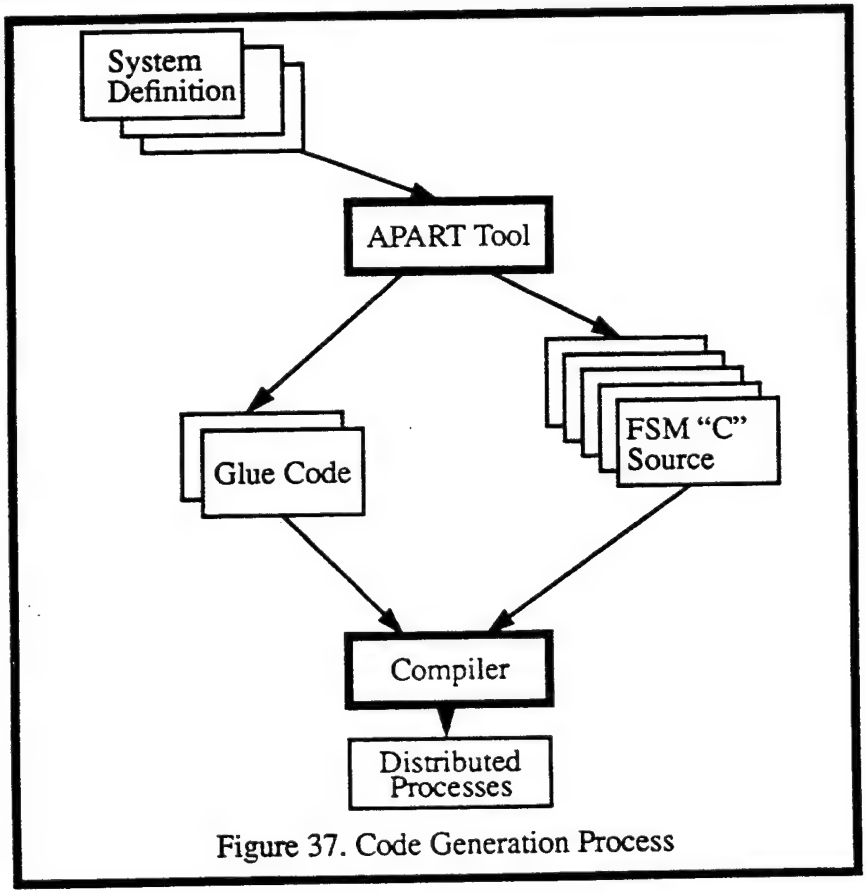


Figure 37. Code Generation Process

The current code generator supports design and implementation of protocols at the application layer.

Compilation in APART is much like other compilers. In this case, the front-end to the compiler is the graphics environment that is presented to the protocol designer. There is an intermediate form of the program, a set of text based files. These files constitute the input to the back end of the compiler. The back end of the compiler is a commercial product called DSET. It converts a text-based program and compiles it to executable applications.

Stage One - Message Collection - In this stage, each transition in each FSM is visited. Each message found is checked against those in the ASN.1 diagram. If it is not there it is added to the ASN.1 diagram with the default type of OCTETSTRING (Empty character string). All variables and their values are also collected together.

Stage Two - Packaging the FSMs for Code Generation - Each FSM is copied into an S-Expression structure for code generation. This structure completely defines the states, transitions, variables, constructors, C-code, queues, and messages for each FSM.

Stage Three - Code Generation Preprocessing - If the protocol designer has requested the simulator to be included, then this step will interweave the FSMs with additional messages. The simulator requires that each state in each FSM update the simulator and wait for its "green light" before transition to its next state. Once stage three is complete, the following stages have no knowledge of the simulator, as all simulator related information has been woven into the protocol.

Stage Four - Application Wide Definition - Generate the text files shared by all FSMs. These include: messages.ad (the ASN.1 messages and their formats), context.ctx (An enumeration of all queues, and messages that are to arrive on those queues), stubs.ac.h (type definitions of global variables, queues, and address constructors), stubs.ac (Commonly used functions, and address constructors for queues), and a Makefile (a shell-like file that knows how to invoke the compilers and linkers with their proper arguments).

Stage Five - FSM Specific Code - Generate the three text files for each FSM. The name of each FSM is used as part of the name for it's three files. If a FSM is named timer, then the files generated are: timer.msl, timer.stubs.ac, and timer.ac. timer.msl is an enumeration of all states in FSM timer that wait on a message, and for each state that receives a message what procedure is to be executed. Timer.stubs.ac contains the functions for all C calls used in FSM timer. timer.stubs.ac will be included into timer.ac automatically. Timer.ac contains the code for all functions referenced in timer.msl. timer.ac generates in-line all constructors for messages. Timer.ac converts all states that have variable value checks in outbound transitions into C "if-then-else" constructs.

The protocol designer must open a command window to the operating system and type, "cd ~/tmp", and "make". This will automatically compile the application and leave an executable in the directory for each FSM.

6.5 Rapid Prototyping

Iterative models of software development are replacing the traditional emphasis on soft-

ware specification. This shift is propelled by the increasing voice of the end user in applications development, and facilitated by prototyping tools.

As competition in the software industry increases, the comfort level of the end user is becoming an important means of stratifying products. Developers, if left alone, tend to get too engrossed in their quest for capability, and often forget the non-expert user. And the preferences of a user are difficult to anticipate. The solution to this dilemma is to bring the user into the development cycle; to modify the product based on user feedback. Traditionally, the cost of several iterations of product development before release, has been an unnecessary expense when one can succeed by "just getting something out the door". As the software industry matures, user expectations for product usefulness and usability increase, thereby driving up the development costs.

Fortunately, software development tools are driving down the cost of developing a prototype. Perhaps the most common example of this is the user interface builder. These packages largely replace old-fashioned graphics coding with drag-and-drop development of user interfaces. The speed and ease of development make it practical to feed user comments back into a series of prototypes, each successively refined from its predecessor.

6.5.1 Rapid Prototyping Protocol with APART

APART is a rapid prototyping tool that allows one to design protocol quickly. Protocol is frequently expressed in terms of messages, queues and finite state machines. APART supports these abstractions in a graphical direct manipulation environment. This enables one to nearly eliminate the (text based) programming normally associated with protocol development by plopping down and wiring together icons. These designs may then be subjected to validation and simulation to iron out bugs early in the development cycle. Further, APART generates C code equivalent to the pictorial designs that are then compiled to executable UNIX applications.

Since protocol engineers can express their cognitive abstraction directly, APART minimizes translation confusion. Without automated translation from design to implementation, there is a risk that good designs are poorly implemented. Further, when problems are found, it may be unclear whether the problem is design, implementation, or related to both. APART's code generation eliminates implementation error, and APART's simulator reflects the problem directly back into the finite state machine design.

APART allows one to concentrate on what is to be communicated, and not how. In the Unix world, one has to be acquainted with the arcane internal calls of TCP/IP in C. Much care has to be taken to ensure that the buffering of I/O is adequate, the interrupt signals wait on the proper vectors, that one message is not lost while processing another, etc. Point to Point communications between two programs require that each side know the others address, or at least they know of a common address where their respective addresses can be stored and forwarded to the other side of the conversation. Protocols with more than two parties especially heighten this concern. APART allows one to ignore this minutia.

The designer can generate a protocol that merely indicates who is to communicate and

when messages are to be sent. This will compile into a working prototype. When ready, the designer can embellish his design to indicate the content of the messages to be sent. This can be compiled as well, for a more functional prototype. Later, when the protocol designer is ready to interface the protocol to an application, he can fill out the stubs files with the appropriate C calls to the API.

APART automatically resolves addressing. Unbeknownst to the protocol designer, there is a name manager that maintains a database of known addresses. Whenever a party to communication starts up, it announces the logical names and physical addresses of its input queues to the name manager. When other parties need to send a message, they query the name manager to obtain the correct address for a given logical name. This has many benefits over compiling the address statically into the code. It allows the various sides of the conversation to run on other machines without recompilation. It allows an indeterminate number of clients in a client/server application. Most importantly, it allows the protocol designer to work with logical names, as the physical addresses will be resolved automatically.

6.5.2 A Prototyping Example: Bob Gets A Date

Consider a trivial example that shows the process of developing protocol in APART. It will also demonstrate the time savings even for a trivial protocol. Note that the diagrams of this protocol represent 13 files (19 pages) of high level "C" source code; code that would otherwise have to be written by hand.

Bob will call Susan and ask for a date. If Susan says yes, Bob prints out "Yippie". If Susan says no, Bob prints out "Rats." Susan will say yes.

This is enough to get started. There will be two parties: Bob and Susan. Bob needs to talk to Susan, Susan needs to talk to Bob.

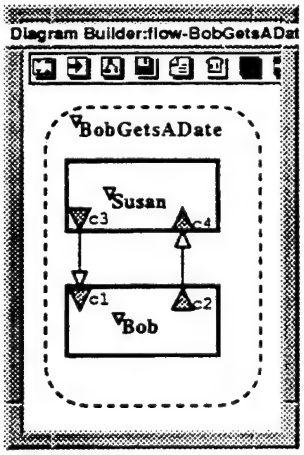


Figure 38. Data Flow Diagram

Notice that the dataflow diagram Figure 38 has no addresses, it merely names the channels of communication and shows where the messages are to be delivered. Addressing will be handled automatically, thus implementing the logical connections pictured.

By clicking on the box Bob, the finite state machine diagram can be opened,(Figure 39). Initially, it is an empty field with gadgets along the top.

As mentioned earlier, Bob will send a message to Susan and ask her out. Then Bob will wait for a response. Depending on her answer, he will print "Yippie", or "Rats."

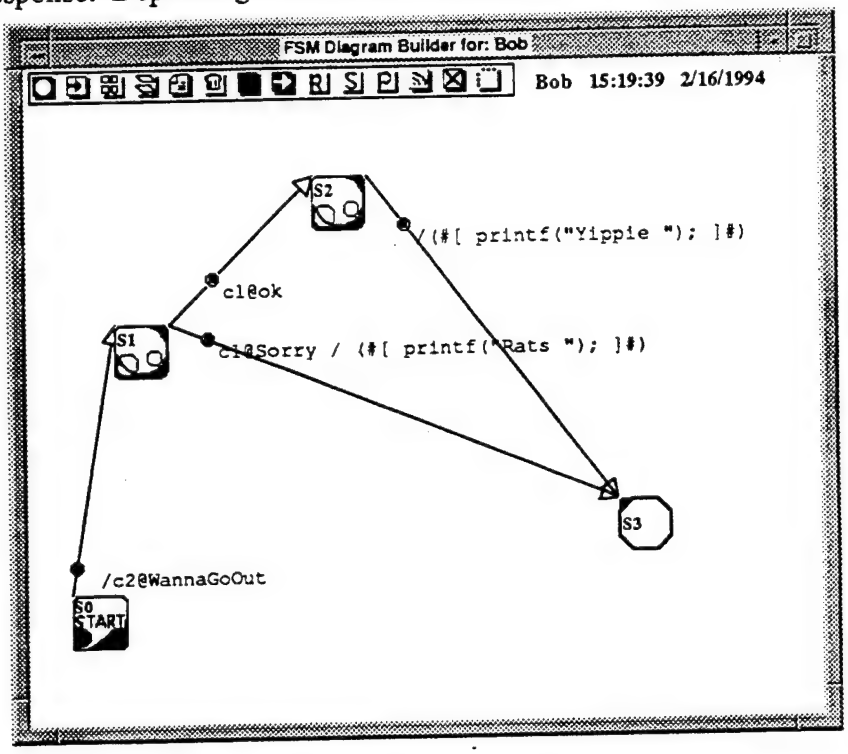


Figure 39. Bob Finite State Machine

S0 is the start state. S3 is the terminate state. On each transition, the precondition is indicated before the slash and the action to be carried out after the slash. The S0 to S1 transition has no precondition (i.e. nothing before the slash). It says, "when in state S0 immediately go to state S1 and send message "WannaGoOut" out the queue called "c2". In state S1, there are two transitions out, indicating two possible paths of execution. The transition from S1 to S2 will be executed if a message called "ok" arrives on queue "c1". If "ok" does arrive, then call the C function printf to print Yippie. However, if the message "Sorry" arrives on queue c1 (while in state s1), then print "Rats". In either case, terminate the program.

Notice also that the messages to be exchanged were declared without specifying the content of those messages. The tool user can fill in the content on a later prototyping pass.

Susan will wait for Bob's call and will answer affirmatively.

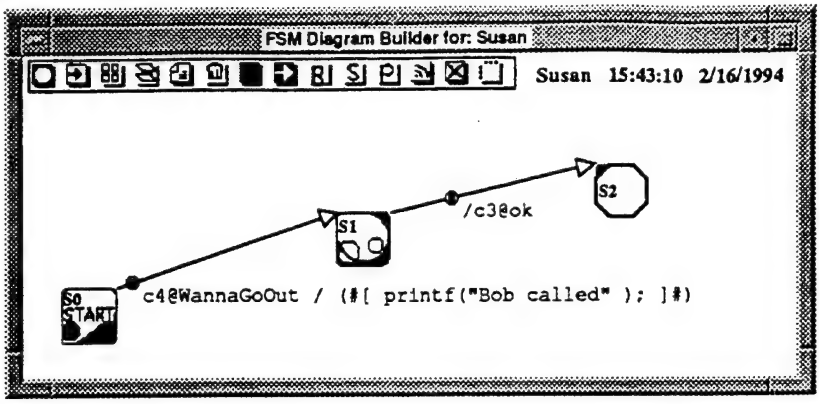


Figure 40. Susan Finite State Machine

This is all that is needed to generate code. Upon code generation, APART will recognize that the messages "WannaGoOut", "ok", and "Sorry", have not been specified. It will construct empty messages for you. It will also add messages needed for its own purposes. The messages can be seen in Figure 41.

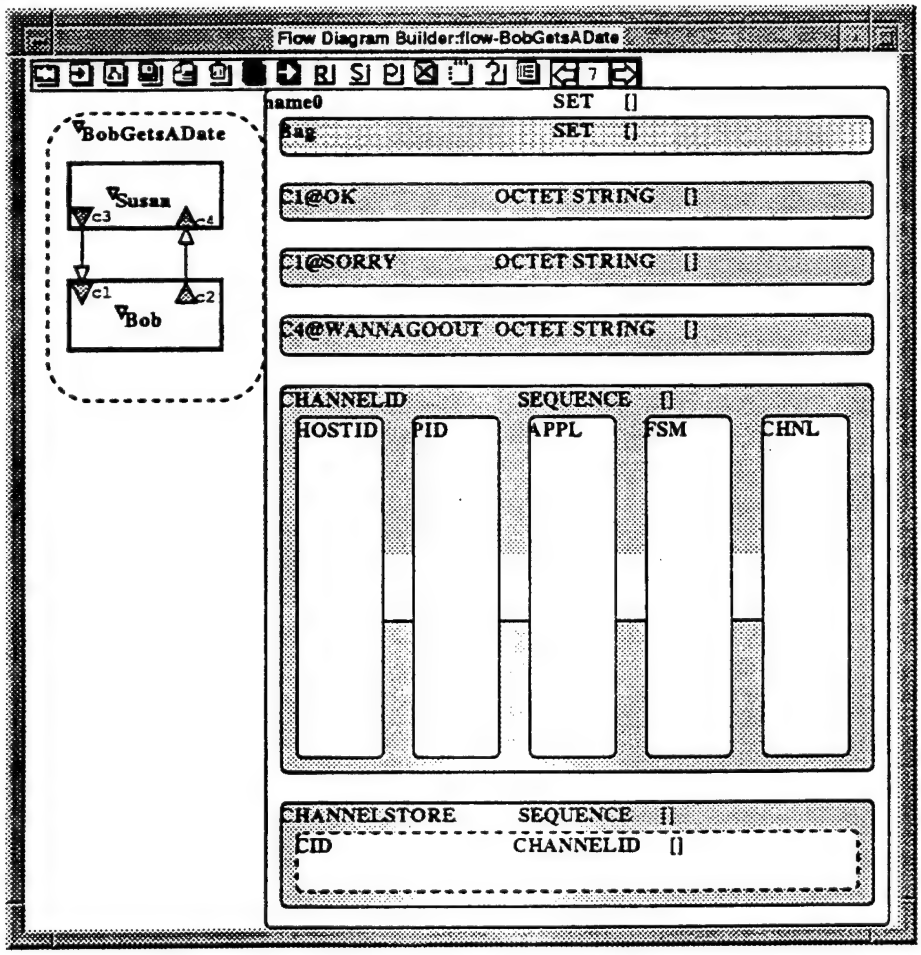


Figure 41. ASN.1 Message Format Diagram

The generated code compiles and runs. Bob and Susan can be run as two tasks on the same computer, two computers on the same LAN, or two computers on opposite sides of the world (if they are both on the Internet).

This finishes a working prototype. The designer can now go back and change the finite state machines, add new finite state machines, or specify the content of the messages and recompile. Each prototyping pass fills out more detail and this iteratively transitions the prototype into the desired protocol.

The point of rapid prototyping is to produce a better product through iterative refinement. Since APART supports the entire design/test/evaluate cycle of software production for protocol; it increases the number of passes one can make on their product. APART communicates to protocol designers in a natural form (graphical finite state machines). It automates addressing. It provides simulation and validation tools for debugging purposes. It generates compilable C code equivalent to the design. A single protocol development pass can take hours or days with APART, instead of weeks or months. This dramatically changes the landscape of development as it becomes cheap to include user feedback in the production of software.

7.0 Groupware

Groupware is a generic term for software that electronically integrates the work of two or more users. It is different from other software applications, in its necessity to unify the work of users on separate computers. A fair percentage of existing groupware applications are little more than constrained mail facilities. However, a lot of the promise of groupware centers around potential applications that require a more complex interaction between computers.

A groupware application must unify the work performed on separate machines, so there must be some "rules of unity" that allow or disallow the inclusion of an individual piece of work into the unified task. Whether acknowledged or not, these rules constitute a protocol, in that there are messages passed back and forth between computers and, in the context of available resources, rules define the course of the interaction between the computers.

The scripting of allowable interaction is a protocol; it is a protocol that is specific to the groupware application. A groupware spreadsheet may not allow two different users to edit the same field at the same time; or if it does, it should assert a priority or otherwise disambiguate the effect upon the data field. The procedural turn taking that must occur between the computers, and the rules that define who wins in such conflicts constitute a protocol. If the protocol is trivial enough, developers may not find it valuable to explicitly acknowledge it as such. However, as groupware applications become more complex, the associated protocols will become more burdensome to debug. By acknowledging the interaction as a protocol, developers can focus abstractly on the problem of the interaction rather than cloud the issue by burying the problem in the details of the implementation.

Groupware developers should heed the lessons of protocol research. The number of possible interactions between N parties utilizing even a simple protocol can be difficult to anticipate. There are some well accepted metrics for good protocol; for example, one would likely want their protocol to be free of deadlock. By incorporating metrics, techniques, and tools from protocol research, groupware developers can minimize their own development effort.

7.1 Protocol Independence

This section will document the merit of explicitly separating the group interaction from the core functionality of a groupware application. Much like user interface design, the design of a group's allowable interaction is a nearly orthogonal problem to that of designing the functionality of an application. The term "protocol independence" was invented by the authors in the tradition of "dialog independence" for user interfaces, and "data independence" for abstract data types. Like user interface design, group interaction should be designed iteratively, for maximum user sensitivity. By separating the group interaction from the rest of the code, one divides a groupware application into two better understood parts: single-user code, and protocol. Figure 42 provides a block diagram for applications that are organized for protocol independence.

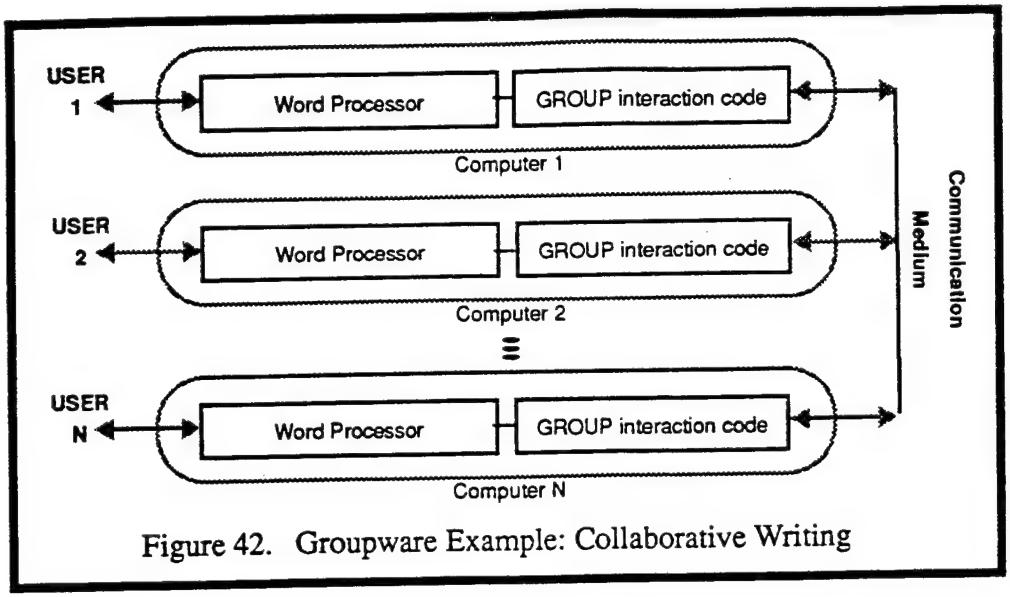


Figure 42. Groupware Example: Collaborative Writing

Methodologies for group interaction that require a heavy emphasis on specification tend to perform poorly; as no matter how well intentioned or informed, one can not always anticipate the needs or preferences of a group of users. As a result, a highly iterative development strategy is recommended. Prototypes of group behavior can be tried and refined as hindrances are discovered. This iterative design methodology is only possible if either, the entire application can be rapidly prototyped, or the protocol is developed independently of the rest of the application.

Protocol independence in a groupware application advocates that a distinction be made between the design of the group interaction and the design of the rest of the application. Protocol Independence:

- separates the complexities of an application with multiple users from traditional single-threaded (i.e. single processor) software.
- allows each portion to be developed and upgraded independently.
- enables the conversion of existing applications to distributed applications; only the group interaction code need be developed.
- enables the conversion of existing applications to distributed systems applications in an efficient manner while minimizing the look and feel changes to the end user.
- increases the likelihood of good design and end product, as the group interaction development can use years of protocol research. For example, one can apply protocol metrics such as deadlock detection.

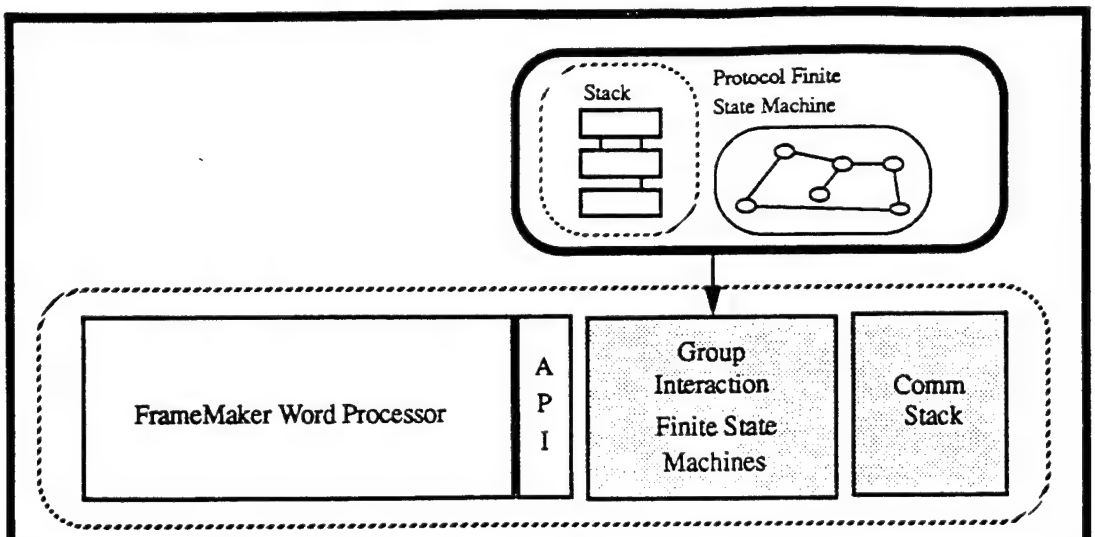


Figure 43. Distributed Application Decomposition

Most importantly, Protocol Independence enables one to use protocol design tools such as APART, in order to get the job done more quickly and cost effectively than with traditional methods. Figure 43 gives a more detailed look at the application architecture under protocol independence. It depicts the APART tool generating the protocol portion of the application.

7.1.1 Group Interaction Is Protocol

Protocol is the allowable interaction of two or more software processes, and this is more difficult to track than uni-processor software. Unless specifically addressed, protocol bugs are more frequent, and often manifested as catastrophic failure.

Fortunately, protocol has objective evaluation metrics, such as: deadlock and livelock. Protocol requires a higher level of program correctness than other software development.

To bundle protocol design with other software development is to reject protocol metrics. Criterion such as deadlock are cast in terms of interactions between processes, the sharing of resources, etc. While any software development theoretically can be cast in these terms, the "full search space" of even a trivial program would preclude the use of techniques such as deadlock detection. By lumping the development of software with objective metrics in with software containing subjective metrics (e.g. Modularity, Top down design, etc.), the lowest common denominator, subjective metrics, shall be applied to the whole. On the other hand, if protocol is designed independently of the rest of the groupware application, one can utilize the tools and techniques of years of research.

7.1.2 Protocol independence limits the effect of reorganization

Protocol Independence, not only speeds protocol development in groupware; it insulates the rest of the application from arbitrary protocol decisions. In user interface development, the principle of dialog independence insulates the user's whims in the interface,

from the rest of the application. In similar fashion, Protocol Independence insulates the organizational structure of the group from the rest of the application.

The topological organization of a group is application, and perhaps user community, specific. If one were constructing an application that routes paperwork to a series of people for their "sign-off", perhaps a pipeline would be most appropriate (e.g. Workflow). Applications where group members collaborate while simultaneously editing the same document may find a star topology sufficient (e.g. Client/Server). Still others applications may find a strict hierarchy more appropriate, for supervisory or propagation efficiency sake. Eventually, as matrix management styles find their way into groupware; adhocracies will allow management structures to evolve and dissolve as circumstances require.

The group topology should be a facet of the group protocol. If one were to switch to a different organization, the protocol would have to be changed to reflect the new command and routing structure. Fortunately, the application specific functionality can remain unchanged. A word processor will still be a word processor, but different people will have the right/responsibility of verifying your new paragraph. By encapsulating the group interaction into the protocol, the rest of the application is unaffected by any reorganization of the group command/control structure.

7.2 APART Produces Groupware

APART is a graphical tool for the design, analysis, simulation, and code generation for protocol. If groupware adheres to the principle of Protocol Independence, then APART can generate the group interaction portion of a groupware application.

APART is a tool which allows a groupware protocol developer to graphically define and manipulate a protocol at a high level of abstraction. From the design, it will create a reference implementation of the groupware protocol and validate the protocol in an interactive environment. If a communication problem is detected at this stage, APART can slow down and snoop the failure condition in simulation mode allowing for several iterations of the design process until the developer is satisfied. APART will generate the code for the groupware application. By speeding the development of good protocol design, APART encourages iterative group-interaction development.

7.3 Various Flavors of Groupware

Groupware can take on various group organizations. The following sections will touch on three: Client/Server, Workflow, and Distributed.

7.3.1 Client/Server Applications

A Client/Server application is a form of groupware in which there are two classes of group members that interact. There are any number of Clients; each user has a Client running on their local machine. A Client represents its user in group interactions. However, there is typically only one server per group. The Server maintains group cohesion, serializes group interaction, delves out group resources, and possibly records group behavior. As groupware applications often have some sort of shared finite resources, the Client/Server model is an obvious and popular technique for resolving group conflict.

The Collaborative Writing Demonstration application, described in the next section, is an example that adheres to the Client/Server model.

7.3.2 Workflow

"Workflow" is a paradigm shift from traditional groupware. Instead of concentrating on the users and how they relate to each other through the sharing of data, Workflow focuses on data, and the users are merely steps on the journey of that data. Workflow is the automation of a well defined (previously paper) process in a group of people.

The prototypical example of Workflow is the insurance claims adjustment process. There is usually a well defined process in an insurance company for handling claims. For instance, a claim arrives, and it is forwarded to the first available adjuster. If the paperwork is incomplete, the adjuster calls the person who made the claim for more information. If the claim meets certain criteria it is automatically accepted and a check is dispensed. However, if there is some unresolved question, the claim is forwarded to someone who is a specialist in that area, and so on. The point of this example is that, if one were to automate the process of routing the information from person to person via computer in the insurance company, one would term this, Workflow.

One metaphor of Workflow is the idea of "in-box"es and "out-box"es. Each person who serves some role in the Workflow process has a set of data items to be processed, an"in-box", and a place to put them after their work is complete, an "out-box". In APART, these are implemented as queues. There are input and output queues and they are specifically designed to store data until it is needed, in the case of an input queue, or until it can be delivered to its destination. Figure 44 shows a data flow diagram for such a Workflow application.

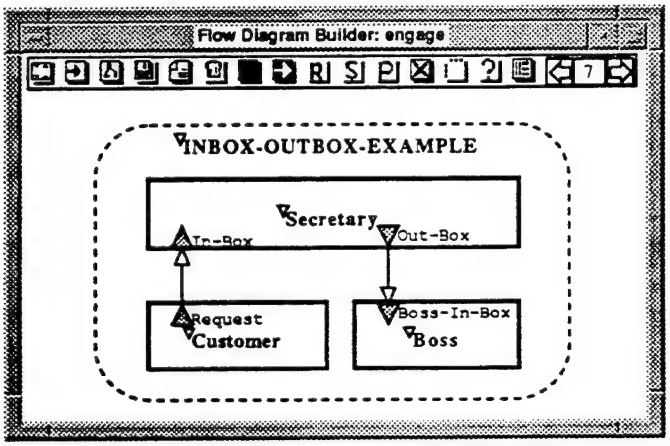


Figure 44. Inbox Outbox Example

Another metaphor is the concept of a pool of resources. For instance, there may be a pool of claims adjusters, any one of which may be able to process a claim. However, it is not known ahead of time, which adjuster will be available. An adjuster will be assigned from the pool, to process a claim at the time the claim arrives.

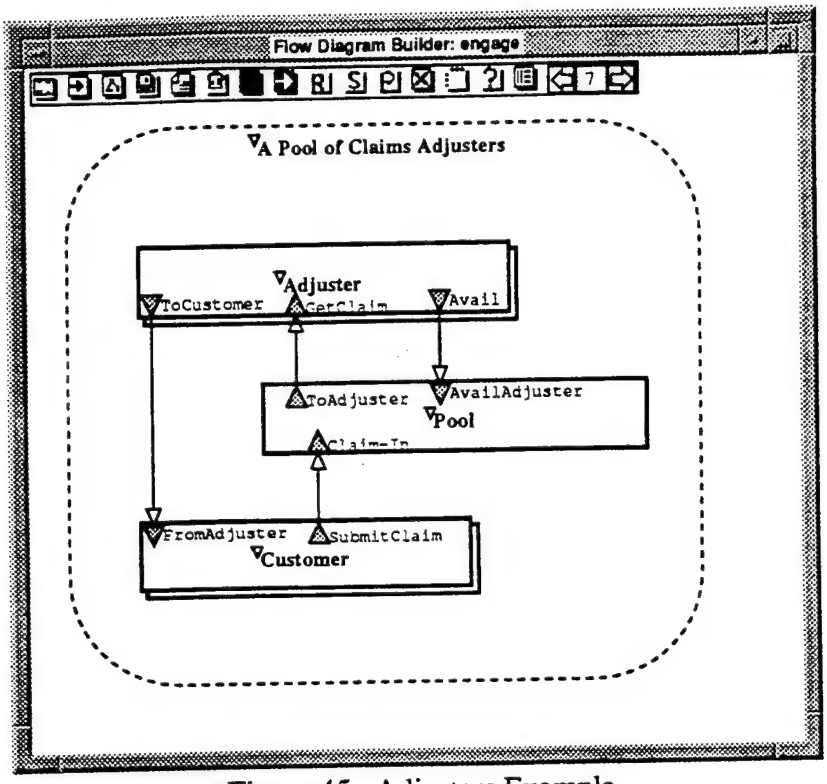


Figure 45. Adjusters Example

Figure 45 show a data flow diagram for an insurance claims example where, adjusters check in to the pool to indicate that they are available. They do this by placing a note in their outbox called, "Avail". When a claim comes in from a customer, it arrives in the inbox, "Claim-In". The address of the customer and the claim are sent to the first adjuster in the inbox "AvailAdjuster", by placing them in the outbox, "ToAdjuster." The adjuster that receives a claim in his inbox "GetClaim", can then process the claim and send it back to the customer by placing it in his outbox, "ToCustomer". The processed claim is delivered to the customer's "FromAdjuster" inbox.

Workflow is very naturally expressed in APART's data flow diagrams. It has been shown that APART allows one to express when and where data will pass from player to player through an organization. Libraries of finite state machines can be built to implement workflow constructs such as pools of resources, and timers. In this way, one can rapidly prototype and generate Workflow applications tailored to the application and organization at hand.

7.3.3 Distributed Applications

Distributed Applications can be used to describe all of Groupware. However, the term tends to imply the application may not have humans as members of the group. In other words, a distributed application may not be constructed for the interaction of humans, but simply a technique for delving out sub-problems to other computers.

One may want to sub-contract out sub-problems to other computers for various reasons. Some computers may have unique capabilities, such as pipeline processors, high speed graphics transformation, database access, etc. Others reasons for distributed applications include speed up through multi-processing, redundancy, or even CPU load balancing.

A non-user based distributed application can have any group architecture. So, APART is a powerful tool to user and non-user based Groupware.

8.0 The Collaborative Writing Demonstration

The Design Specification Document stipulated that an application interface would be devised and delivered as part of the research. The purpose of this deliverable is to demonstrate how protocol implemented with APART can provide service to an application.

In this section, the author will retrace the process of creating the Collaborative Writing application and application interface using APART. The Collaborative Writing Demonstration is a groupware application constructed from a commercially available single user word processor. This distributed application will be structured in accordance with the protocol independence architecture as depicted in Figure 46.

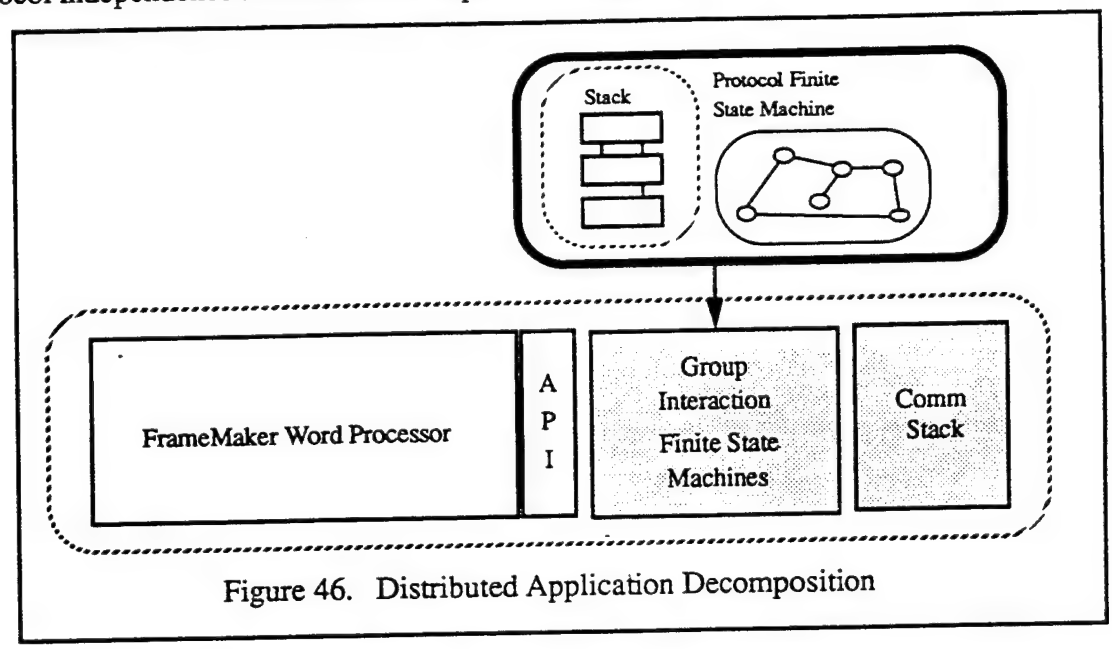


Figure 46. Distributed Application Decomposition

8.1 Walk-through (with protocol rules version 1)

By walking the reader through the steps of constructing the Collaborative Writing Demonstration, one should glean both the level of effort required to upgrade single user software to groupware, as well as the methodology of development. An integral part of this methodology is the design and construction of the protocol to application interface.

Step 1: Define the Goals

Clearly stated goals dramatically increase the probability of productive development. In an effort to showcase the ITKM staff development effort, the Collaborative Writing Demonstration had the following goals:

- Upgrade a commercial word processor to Groupware, using the APART tool.
- Minimize the changes to the end user, so that Collaborative Writing is little different than single-user writing.
- Allow one to work on a private copy of text, and then allow a user to share it with the others only when comfortable.

Step 2: Design the Organization of the Group

It seemed reasonable to assign all users to equal status in the group. In order to simplify serialization and resolve conflicts over resources, the team chose to have an autonomous group moderator. This constituted a Client/Server arrangement, where each user is assigned a client and the autonomous moderator is the server.

Step 3: Define the Group Interaction

A locking scheme was chosen to coordinate the work of the group members. Much like a library, one can check out a book, and others can not use it until it is checked back in. More specifically there is a document server and any number of clients. Figure 47 shows the distributed application organization with one server task and two client tasks. The Finite State Machine tasks are represented as filled in rectangles. Each group member has a client FSM associated with it. Initially all group members have a read-only version of the document. The local editor that holds a portion of the document is represented as the rounded corner object. The document is divided into sections typically on chapter boundaries. The beginning of each section is identified with a hypertext button. The button is represented as a small empty rectangle in Figure 47. The actual look&feel of the word processor can be see in Figure 48. The hypertext button in the read-only document is represented as a "pencil" icon.

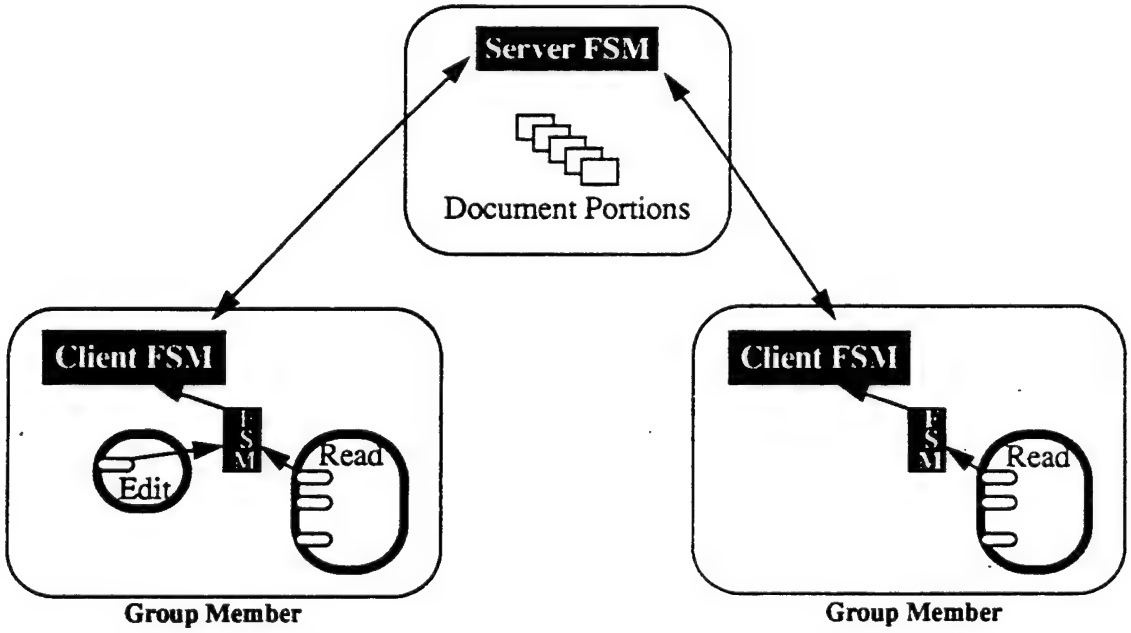


Figure 47. Client/Server Task Organization

Whenever one wants to edit a chapter, one clicks on the hyper text button. This sends a message to the server requesting permission to edit the chapter. If no one else is already working on that chapter, then the server sends the chapter to that client with full read/write privileges. There the group member can use the word processor to change the chapter as

needed. When the group member feels it is in a form sufficient for others to read. Then he presses an "ink-pen" hyper text button to publish it. This has the effect to transmit the new chapter to the server. The server marks it as available for others to edit and then distributes the entire document (with the new chapter) to all group members in read-only form. This new version replaces the previous version on the screen. In this fashion, the group members can work on different parts of a single document individually, and as others publish, they can receive the latest-and greatest, without stepping on each others toes.

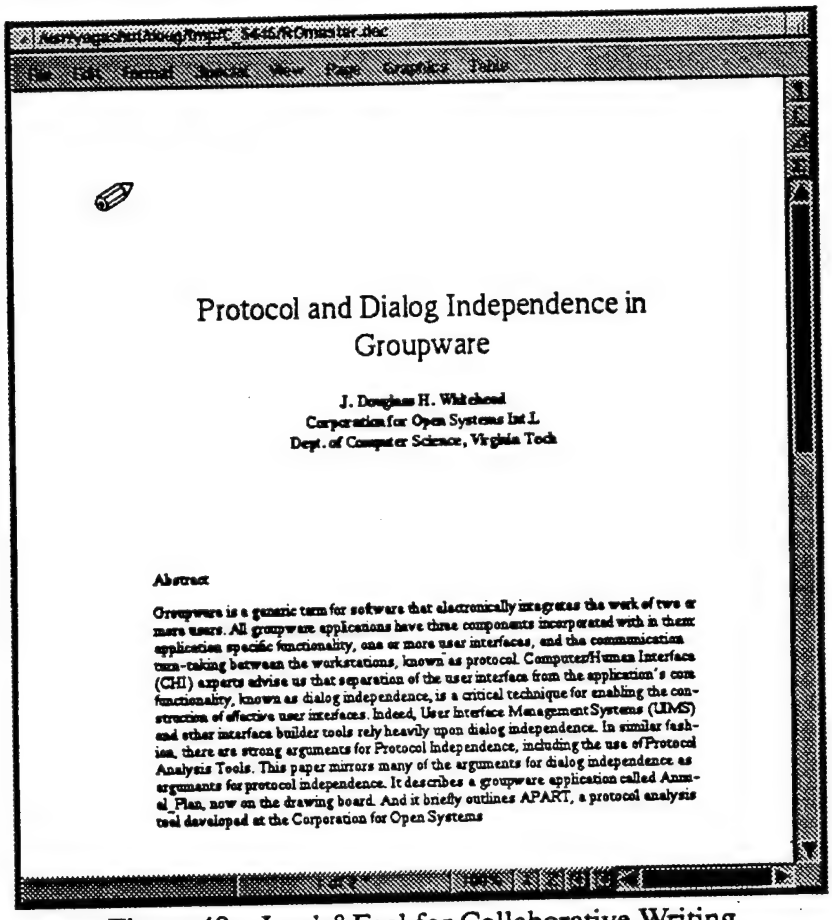


Figure 48. Look&Feel for Collaborative Writing

Step 4: Define the API

In order to construct this demonstration, it was necessary to confirm that the word processor had a bi-directional API. This is a communication mechanism between the word processor and the group interaction code (i.e. protocol) that would be developed with APART.

The word processor would run on each group member's computer as a Unix task. Simultaneously executing on each computer would be the protocol task. The protocol task would be generated by the protocol design tool APART. However, some means of communication between the word processor and the group interaction is necessary. This constitutes the application interface(API).

Communication from the word processor to the Client/Server tasks will be know as group activation. The user needs to be able to request exclusive rights to edit a portion of the document. Later, the user needs to be able to publish the portion and release rights.

Communication from the Client task to the word processor will be know as local activation. The protocol needs to be able to control the word processor in order to open a document, close a document, and build a read-only document from a set of separate portions.

Step 5: Select a Product to Upgrade to Groupware

Often this step is earlier in the development process, but since our goal was to demonstrate the effectiveness of APART; the specific product selected was less important. FrameMaker is a word processor on Unix machines. It has extensive text, graphics, page-layout, and hypertext capabilities. FrameMaker was selected because it happened to be available at the time, it's hypertext, and programmatic API.

Step 6: Design the Dataflow

Based on earlier decisions, there are any number of users all of equal status. Each user has a client associated with it; or restating this more correctly, each client is the protocol for a user in the group. There is only one server, and it is capable of talking to and listening from any client. The server must also maintain a list of active clients. A button task will be executed each time someone clicks on a hypertext button. The button task will inform the client task that the user wants something. This organization is reflected in the dataflow diagram of Figure 49.

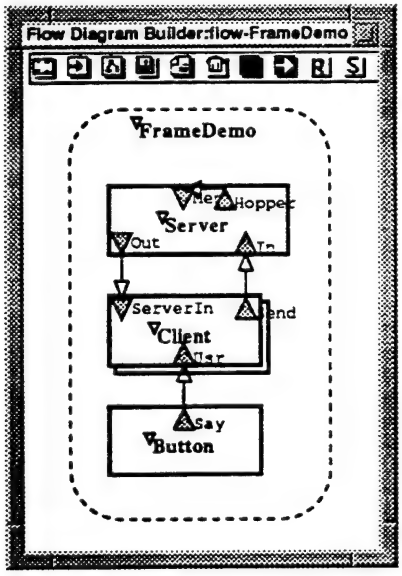


Figure 49. Collaborative Writing Data Flow

FSM Button informs the client of the user's request on output channel "Say", which will be delivered to the Client's channel "Usr". Clients have channels to and from the Server. Whenever a client checks into the server, the client's address will be thrown into a list of active clients (e.g. the "Hopper").

Step 7: Design the Finite State Machines

The Client finite state machine diagram is shown in Figure 50. After requesting and receiving membership the FSM will be in the normal wait state S6. If the user requests to edit a portion of the document, then from S7 the FSM can either Open the portion or report that it is not available. If the user is finished with a portion, then S9 passes on the published portion to the server. If server indicates that there is a new read-only document, S10 (the open book icon) handles closing the old document, and opening the new one. In any case, execution returns to state S6.

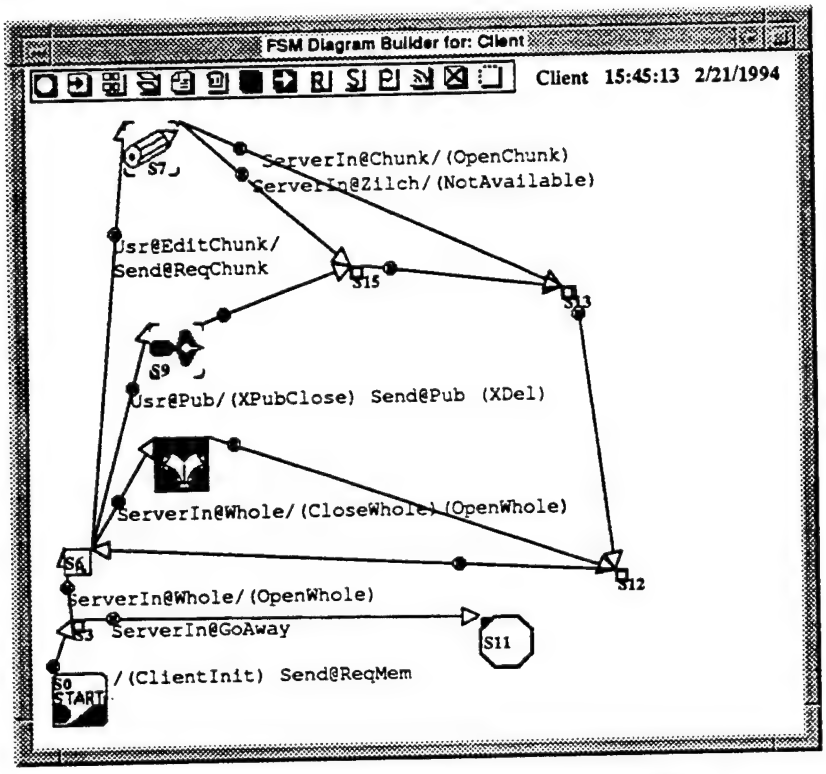


Figure 50. Collaborative Writing Client FSM

The Server finite state machine diagram is shown in Figure 51. S4 is the normal wait state. States S6, S7, and S15 add new Clients to the list of Clients, if they are not already on the list. State S11 handles a request for a portion of the document, and returns a message of either "Chunk" or "Zilch" to indicate whether the portion is available. When a portion is returned, S13 will mail a new read-only document to each member on the client list.

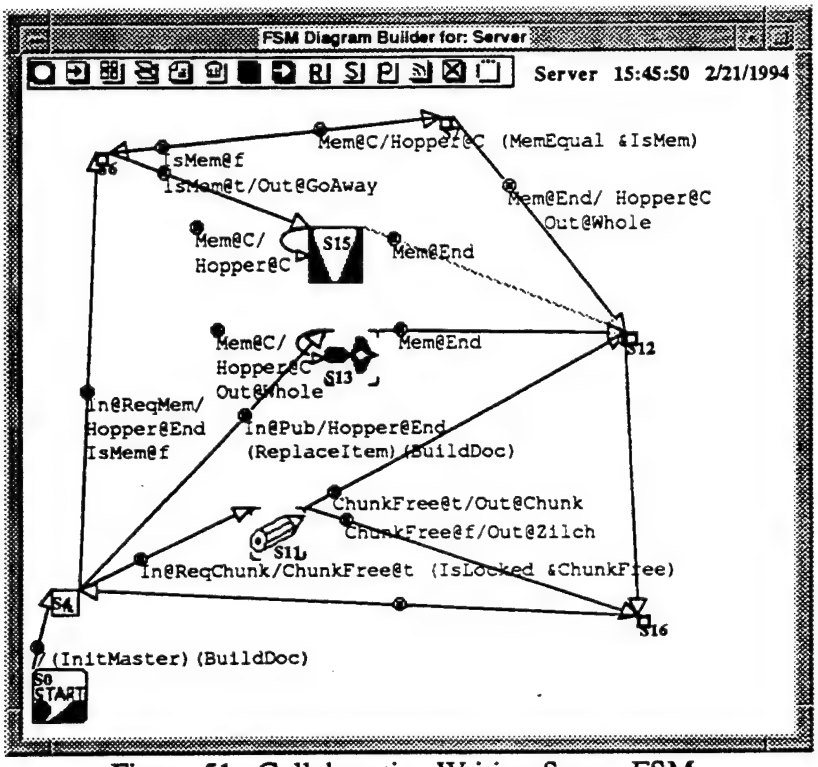


Figure 51. Collaborative Writing Server FSM

The Button finite state machine is shown in Figure 52. There are two messages the user can send to the client with the Button FSM, EditChunk or Pub. This FSM scans the command line for an address and uses it to mail the message to the user's client. The Button FSM is so trivial, that the author chose to put the C code for command line parsing directly on the FSM diagram, rather than hiding it in a stubs file.

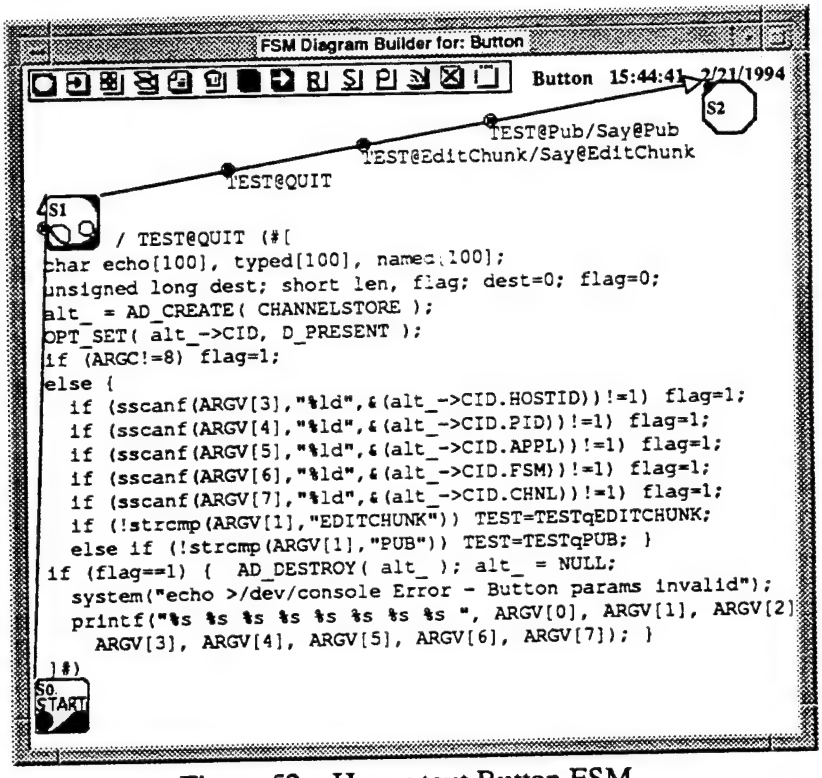


Figure 52. Hyper-text Button FSM

Step 8: Generate Code, Compile, and Debug the Skeletal Protocol.

The design should generate text source code without additional information. Each message found in the FSM diagrams will be placed in the ASN.1 diagram (if it is not already there). Each FSM diagram will generate a file of "stub" procedures that create a place holder for API calls, to be filled in later. The most frequent design bugs caught at this stage include: miss-typed channel names (as indicated on the data flow diagram), accidentally disconnecting one end of a transition, forgetting to indicate start or terminate states, etc.

Type "make debug" in a command window to compile it with debugging messages included. Execute the code, it should exchange messages at this point, but not interact with the word processor, nor pass data in the messages.

Step 9: Complete the Messages

Some messages, such as requests or acknowledgments, need no content. Others need to pass simple text strings or integers. Others pass whole files as large Octet Strings.

Step 10: Program the Constructors

By shift-right mouse clicking on an outgoing message, one can edit the constructor for that message as seen in Figure 53. This is C code for filling in the content of the message about to be mailed. "msg" is the pointer to the message to be filled in. Lines 1 and 2 construct a character string of a filename. Line 3 allocates space in the outgoing message for a character string field called NAME. Line 4 copies the character string from the last message arrived field called NAME into the outgoing messages field also called NAME. Line 5 allocates space for a file in the outgoing message field called FILE1. bufffromfile on Line 5 is a procedure that copies the contents of a file into the space.

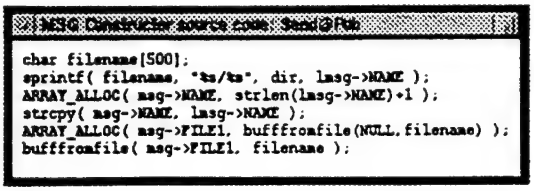


Figure 53. Constructor: Put Document in Message

Step 11: Generate Code, Compile, and Debug the Protocol.

The design should generate code for the protocol with complete messages. The most common design bug found at this stage is a message field miss-spelling, or type miss-match between the ASN.1 diagram and the constructors.

Step 12: Connect the Protocol to the Application (application interface)

Local activation is the set of procedures that implement the ability of the group to wake up the local word processor. Local activation was implemented by modifying an API for FrameMaker called "docclient". Essentially docclient is a back-door for programmatic synthesis of key clicks and dialog interaction. Local activation for the collaborative writing demonstration consisted of code to implement three functions.

- "opendoc" opens a file in framemaker in either normal or read-only document, and with or without a hypertext publish button.
- "closedoc" saves an opened document and dismisses the window.
- "builddoc" appends several files to construct a single document with hypertext edit buttons for each file portion.

Figure 54 shows a portion of the C code that implements "opendoc". It consists primarily of stuffing predefined character strings down an i/o stream to tell FrameMaker what to do.

Group activation is the set of procedures that implement the ability of a user on the local word processor to wake up the group. Group activation was implemented as hyper-text buttons. In FrameMaker, one can specify a Unix command to be issued whenever a hypertext button is clicked. By constructing a Button FSM that sends a message to the client when it is executed, the hyper-text button can execute the Button FSM, and thus notify the client FSM of the user's desire. Group activation for the collaborative writing demonstration consisted of the following two functions.

The Button finite state machine is shown in Figure 52. There are two messages the user can send to the client with the Button FSM, EditChunk or Pub. This FSM scans the command line for an address and uses it to mail the message to the user's client. The Button FSM is so trivial, that the author chose to put the C code for command line parsing directly on the FSM diagram, rather than hiding it in a stubs file.

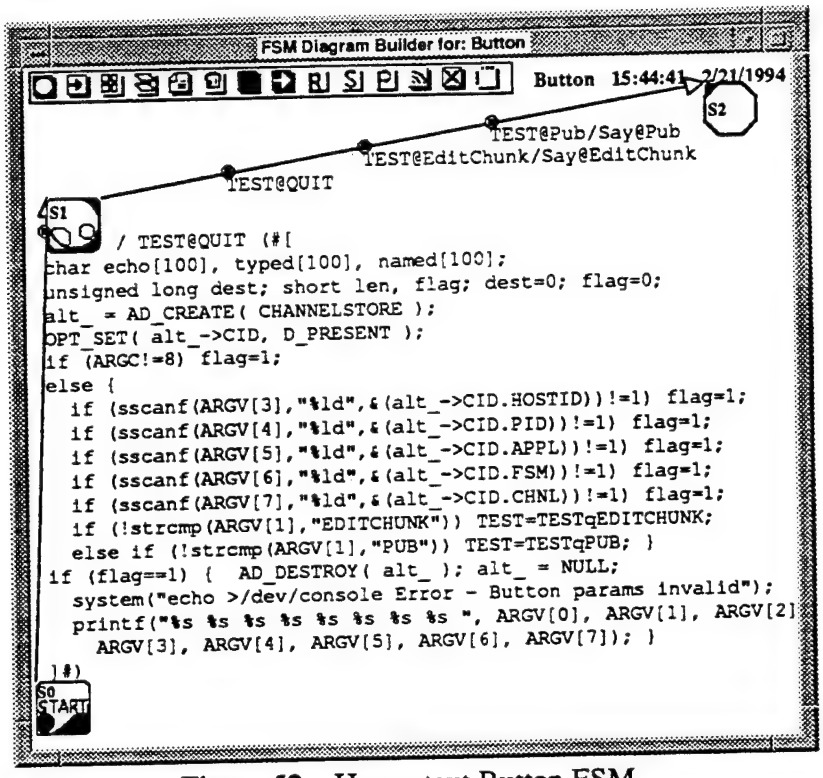


Figure 52. Hyper-text Button FSM

Step 8: Generate Code, Compile, and Debug the Skeletal Protocol.

The design should generate text source code without additional information. Each message found in the FSM diagrams will be placed in the ASN.1 diagram (if it is not already there). Each FSM diagram will generate a file of "stub" procedures that create a place holder for API calls, to be filled in later. The most frequent design bugs caught at this stage include: miss-typed channel names (as indicated on the data flow diagram), accidentally disconnecting one end of a transition, forgetting to indicate start or terminate states, etc.

Type "make debug" in a command window to compile it with debugging messages included. Execute the code, it should exchange messages at this point, but not interact with the word processor, nor pass data in the messages.

Step 9: Complete the Messages

Some messages, such as requests or acknowledgments, need no content. Others need to pass simple text strings or integers. Others pass whole files as large Octet Strings.

Step 10: Program the Constructors

By shift-right mouse clicking on an outgoing message, one can edit the constructor for that message as seen in Figure 53. This is C code for filling in the content of the message about to be mailed. "msg" is the pointer to the message to be filled in. Lines 1 and 2 construct a character string of a filename. Line 3 allocates space in the outgoing message for a character string field called NAME. Line 4 copies the character string from the last message arrived field called NAME into the outgoing messages field also called NAME. Line 5 allocates space for a file in the outgoing message field called FILE1. bufffromfile on Line 5 is a procedure that copies the contents of a file into the space.

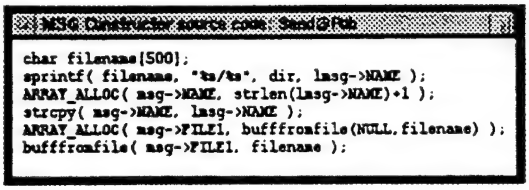


Figure 53. Constructor: Put Document in Message

Step 11: Generate Code, Compile, and Debug the Protocol.

The design should generate code for the protocol with complete messages. The most common design bug found at this stage is a message field miss-spelling, or type miss-match between the ASN.1 diagram and the constructors.

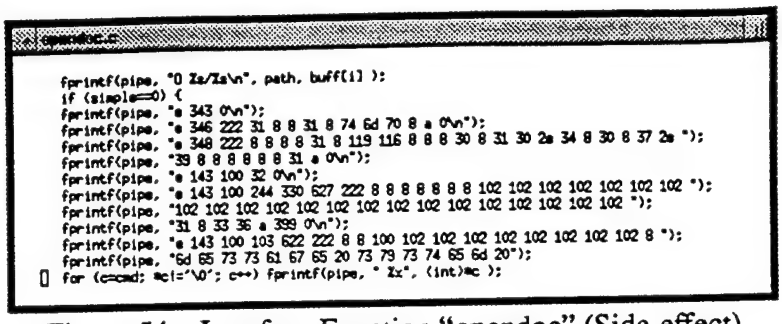
Step 12: Connect the Protocol to the Application (application interface)

Local activation is the set of procedures that implement the ability of the group to wake up the local word processor. Local activation was implemented by modifying an API for FrameMaker called "docclient". Essentially docclient is a back-door for programmatic synthesis of key clicks and dialog interaction. Local activation for the collaborative writing demonstration consisted of code to implement three functions.

- "opendoc" opens a file in framemaker in either normal or read-only document, and with or without a hypertext publish button.
- "closedoc" saves an opened document and dismisses the window.
- "builddoc" appends several files to construct a single document with hypertext edit buttons for each file portion.

Figure 54 shows a portion of the C code that implements "opendoc". It consists primarily of stuffing predefined character strings down an i/o stream to tell FrameMaker what to do.

Group activation is the set of procedures that implement the ability of a user on the local word processor to wake up the group. Group activation was implemented as hyper-text buttons. In FrameMaker, one can specify a Unix command to be issued whenever a hypertext button is clicked. By constructing a Button FSM that sends a message to the client when it is executed, the hyper-text button can execute the Button FSM, and thus notify the client FSM of the user's desire. Group activation for the collaborative writing demonstration consisted of the following two functions.



```
fprintf(pipe, "0 %s/%s\n", path, buff[i] );
if (simple==0) {
fprintf(pipe, "e 343 0\n");
fprintf(pipe, "e 346 222 31 8 8 31 8 74 6d 70 8 e 0\n");
fprintf(pipe, "e 348 222 8 8 8 8 31 8 119 116 8 8 8 30 8 31 30 2e 34 8 30 8 37 2e ");
fprintf(pipe, "39 8 8 8 8 8 8 31 e 0\n");
fprintf(pipe, "e 143 100 32 0\n");
fprintf(pipe, "e 143 100 244 330 627 222 8 8 8 8 8 8 8 8 102 102 102 102 102 102 102 ");
fprintf(pipe, "102 102 102 102 102 102 102 102 102 102 102 102 102 102 102 ");
fprintf(pipe, "31 8 33 36 e 399 0\n");
fprintf(pipe, "e 143 100 103 622 222 8 8 100 102 102 102 102 102 102 102 102 8 ");
fprintf(pipe, "6d 65 73 73 61 67 65 20 73 79 73 74 65 6d 20");
for (c=cmd; *c!='\0'; c++) fprintf(pipe, " %x", (int)*c );
```

Figure 54. Interface Function "opendoc" (Side-effect)

- "editchunk" would indicate the user wants to edit a specific portion of the document.
- "publishchunk" would indicate that the user is finished with a portion of the document and no longer needs it.

Figure 55 shows a hypertext button and its definition. When one clicks on the hypertext button, the Unix command "Button EDITCHUNK g1 98 69 188 1 0" will be issued. "Button" is the small FSM created just for group activation; all it does is construct a message out of the items on the command line and sends it to the Client FSM. EDITCHUNK indicates that the user wants to edit a portion of the document. "g1" is the name of the document portion. "98 69 188 1 0" is the address of the client. When the Client issued the "opendoc" for this file, it substituted its own address into the hypertext buttons to insure that it receives the user's requests.

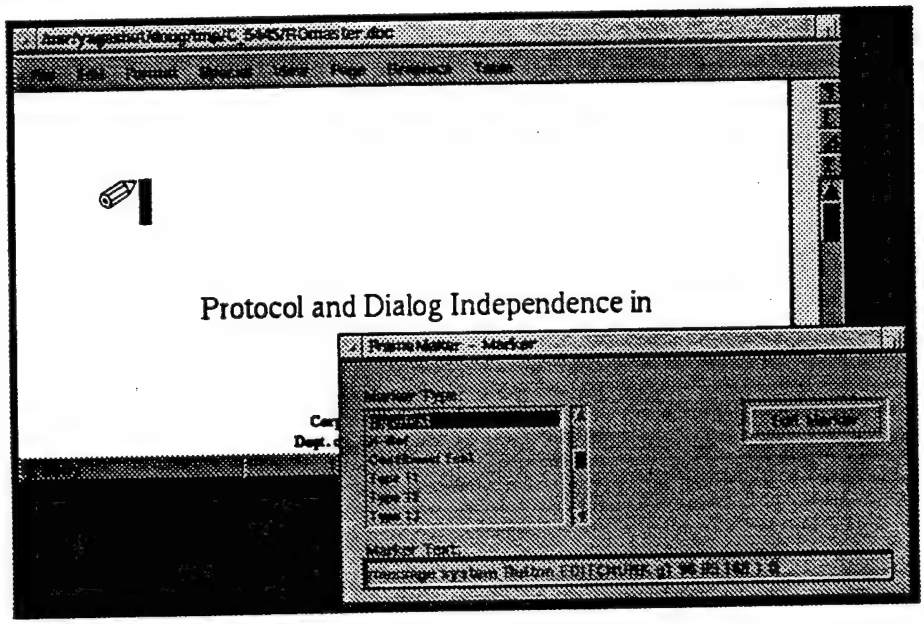


Figure 55. Hypertext Button Definition

Now that the bi-directional API is implemented, as a set of functions, source code for any side-effect functions, in stub files, can be filled out.

Step 13: Generate Code, Compile, and Debug the Groupware Application

This completes the prototype of the groupware application.

Step 14: Community Usability Testing

The end users should be allowed to try out the application. Undoubtedly, they will have suggestions for improvement. Based on feedback and cost/benefit analysis, one may go back to a previous Step with new insights. Fortunately APART makes the modification of designs inexpensive, to allow multiple iterations to hone in on an acceptable groupware application.

Suggestions received during the construction of the collaborative writing demonstration included the following items:

- A more direct API than docclient. By obtaining information directly from the developers of FrameMaker, Consider reducing the "simulated mouse clicks" and screen flashes.
- A group archive capability. The Server FSM could be augmented to save the document after each publication to prevent both catastrophic failure, and loss of information through deletion.
- Edit rights time-out. Currently, once one checks out the rights to edit a portion, he has exclusive rights until the changes are published. Time-out should be easy to implemented at the Server. However, one must then resolve the conflict between the two sets of changes to a portion (i.e. those that result from a timed-out session, and those from the member who obtained rights to change it after the time out).
- Hierarchical allocation of editing rights. One may want to hand out responsibility for major chapters to designated individuals, who then delve out sub-chapters to their group members, etc. recursively. This would suggest that the group be organized as a set of groups, instead of a static Client/Server model. This would also require that a document be able to be infinitely divisible into sub-portions.
- Access protection. Presumably different people may have different authority for group membership, read, or editing rights. This would probably be implemented as some sort of database accessible by the Server FSM.

8.2 Alternate Protocol Rules (version 2)

Another protocol was developed in parallel to the "locking" mechanism described in this chapter. It was also a Client/Server model, but it allowed any number of group members to check out and edit the same document portion simultaneously. It resolved the conflicts with a version control technique.

Each document portion has an associated version number that was sent to the clients along with the document chunk. When a portion was published it would be either appended to the existing portion held by the Server or replaced. If the portion to be published had a version number greater then or equal to the version held by the Server then it would be replaced. Following the insertion or replacement the Server would increment the version number.

In this way, the document would include all comments, and allow one to check them out and merge them together over time. This effort used the same API as the locking protocol; only the protocol design differed. This illustrates the power of protocol independence. Once the protocol designer understood and implemented the API, it was possible to design and redesign the protocol independently of the application.

It was demonstrated that APART allows one to take an off-the-shelf, single-user, product and create an integrated multi-user product out of it. The modification to the product's user interface was minimal, just two hypertext buttons. In essence utilized a back door, invisible to the end user, in order to maintain group coordination. This potential of this technology for enterprise integration is vast; as it allows the coordination of legacy systems into a new cooperative whole.

9.0 Usability Testing

To accomplish the usability testing task, a separate document was produced titled "The APART Protocol Development Environment Test Plan." The goal of the test plan was to determine the usability of the APART Protocol Development Environment Tool by the average, rather then the expert user. The emphasis was on the core features and basic building blocks in the development of a simple application.

The focus is on how quickly and easily a new, but qualified, user is able to learn to use the APART tool and produce a simple application following specific instructions. Success was achieved if the tester could become familiar enough with the tool to independently enhance and modify the application. Testing consisted of four steps:

1. Automated introduction to the tool and the answering any questions the tester may have at the start.
2. Copying exercise to see if the tester could master the basics of the tool interface.
3. Simple enhancement to see if tester could deal with diagram edit functions of tool and simple understanding of diagram information.
4. More difficult enhancement of protocol to see of tester had a good understanding of protocol being represented by tool diagrams.

An attempt was made to keep the testing environment constant for all testers. To achieve this goal a playback script was constructed for the tool which automated the demonstrated of the construction of a two party protocol. The tester was expected to repeat the same or similar sequence of activities in constructing the demonstration protocol. This phase of testing accomplished two things. The team was able to observe the level of ease or difficulty in using the basic interface functions of keyboard and mouse actions along with graphical feedback. Both observations and any questions that the tester asked were recorded. This test activity also provided training in the basic use of APART. The team was trying to determine at a coarse level just how much training was necessary for a new user to get started.

9.1 Testing Results Summary

The testers were provided the test plan ahead of time and requested to read through it. The playback demonstration took approximately ten minutes with about five more minutes to answer questions. With this fairly minimal amount of training all testers were able to use the tool and build the demonstration protocol. The team was quite pleased with this outcome. The fact that first time users could successfully operate the tool and comprehend the graphical information was a very positive result. I didn't expect the testers to be able to complete the second round of protocol modifications. The fact that they did was a pleasant surprise. I believe it indicates that the APART tool is on the right track.

A fair amount of effort went into the test plan. The test plan contained a detailed script with graphic examples. It acted as a training aid . Something not contemplated early on. I suspect that the quality of the test plan played a part in the success of the testing.

9.2 Testing observations

A few interface nuances were observed or elicited questions which pointed to things that could be cleaned up. The following is a list of features that were identified as things that could be either cleaned up in the tool code or clarified in documentation:

- In data flow diagrams, it is easy to accidently click on channel connectors which cause a duplicate channel to be added to the diagram. Once the user is aware that this can happen it is fairly easy to avoided. It is possible to change the mouse-key assignment to make this less likely.
- Some functions have a standard dialog box which requires the user to click on 'apply' or 'ok' buttons. The difference between these buttons needs to be explained. Some form of automated help could avoid reference to paper documentation.
- The hot spot for the cursor is not always obvious. Once the user is aware of what is the hot spot this problem goes away, but is could be improved by a more overt hot spot graphic.
- It is possible to make a stack object with zero size if the user accidently clicks a couple of times while in make-stack mode. Its rare that a user would do such a thing, but it could be avoided by having the tool automatically discard stack objects at creation time with very small size.
- Each diagram has many different text objects which could be the focal point for keyboard input. When in input mode the cursor changes shape to a small vertical line. Its perhaps to small and sometimes hard to find in complex diagrams. Either a larger cursor for edit mode or some additional feedback graphic would be useful to improve status feedback.
- When drawing transition arrows in FSM diagrams a new user can get confused as to which end point, source or destination, the next click will produce. This can be cleared up by enhancing the graphical feedback to signal a source or destination.
- In order to enter edit mode of empty pre-conditions, the user needs to know that the hot spot is just to the right and down from the transition anchor point. Once the user knows were to point this is no longer a problem. It can be improved by supporting clicking on the anchor point itself to enter edit mode for preconditions.
- The preconditions and side-effects have a simple syntax that is explained in paper documentation. Some on line help could eliminate the need for paper documentation.
- The parsing of preconditions and side-effects do not recover well from syntax errors. The syntax is so simple this tends not to be a problem for experienced users, but this could be improved for first time users.
- In FSM diagrams to call up a constructor window the text cursor must be pointing somewhere within the text string of the output message name. Putting it at the end of the string does not work. Once the user is aware of this fact the problem goes

away, but the hyper-text button behavior could be improved to work on the end of string as well.

- In message format diagrams, objects can be organized horizontally or vertically. This is for visual convenience only and has no semantic meaning. This is explained in documentation. On-line help could benefit first time users.
- Its not intuitively obvious that the user only has to enter message format names for the input side and the tool will match up the outgoing messages by the same message name. This is explained in the documentation, but could also be stated in on-line help.
- Depending on the users background, some training on how to read the diagrams is necessary. This is simply a statement of fact. Nothing needs to be done.
- Some errors are placed in dialog boxes which are obvious. Some errors show up in the command window where the tool was started. This is covered in documentation, but could be pointed out in on-line help.
- Pop-up menus in flow diagrams are created in real time and are slower then other menus. This could be optimized for performance.
- The Meta key is not labeled on the keyboard. This could be added to the on-line help.
- Flow-diagram pop-down menus can be dragged after opening. The dragging behavior should be removed.
- The current version of the Tool uses "C" code as the development language for the side-effects and glue code. As a result all "C" restrictions such as reserved words are limitations inherited in the diagrams. No action to remove this fact is planned.
- Two reserved words are used in the language: 'lmsg' for last message received and 'msg' for current message under construction. This could be pointed out in on-line help.
- In the flow diagrams, tool functions are selected by clicking on what appears to be text objects. It would be a little more obvious for first time users to make them look more like buttons.
- In format diagrams, horizontal is the default. Some users may want vertical as default. This could be made a configuration option.
- In editing format diagrams, when a item is added for the first time the parent object's type is automatically changed form primitive to constructed. The reverse is not true. This could be added for improved convenience.
- Octet-String is a common type and the default. For convenience is should be added to the options menu.
- It would be a little cleaner to remove the 'OK' button from the dialog box that reminds the user to click on the stack to be saved.
- Flow-diagrams and format diagrams are in the same window. The user needs to change mode when going from one diagram to the other for editing. This can be

made a little more convent if the triangle used to enter format mode acted as a toggle with corresponding feedback. The cursor does change as feedback to which mode is active, but additional feedback could be helpful.

9.3 Possible Enhancements

The Tool currently supports modularity by dividing a protocol stack into finite-state-machines. In the current version, all states in a single fsm diagram are at the same viewing level. This was based on the observation that most protocol FSMs have 30 or fewer states which could reasonably be viewed all at once. To support FSMs with very large state space some form of graphic macro capability could be employed. This would allow a group of states and transitions to be packages and graphically shrunk to a single icon. This is similar to the concept that was used in shrinking objects in the message format diagrams.

Appendix : Protocol Decomposition

1 Protocol Decomposition

Object oriented design typically includes the definition of a set of objects which correspond to the real world of the problem domain. In our case the real world is data communication protocol and how people have dealt with protocol through open system standards. A worthwhile exercise was to enumerated the objects in the real world domain, and to identify corresponding objects within APART. To help discover the working set of objects, the team performed two exercises. The first was to review a protocol specification and list all the generic things which are described in the specification. T he resulting list can be found in subsection 2. In addition an example state transition was scrutinized and all of its steps recorded. This recorded list can be found subsection 3. From these two lists, an initial set of objects or protocol building blocks have been formulated.

From the exercises and past experience, the initial building block design is based on the assumption that standard protocol of the upper layers can be decomposed into the following items:

1. Finite State machine
2. Data flow diagram
3. Message format diagrams
4. Set of protocol actions which can be constructed from a library of function parts.
5. State variables and predicates
6. Events, Timers, and Queues
7. A high level notation which can handle the large majority of specifying the variables, predicates, and actions.

One of our research hypotheses is that protocol can be decomposed into the above logical parts and that the parts will be sufficiently generic to be reused in the specification of multiple protocols. For a tool to be able to construct an implementation, additional generic parts will be needed. Each generic part will be handled through some aspect of the tool's graphic interface.

2 Components of a Protocol

This subsection is the result of an exercise to discover what is needed for protocol building block objects within APART.

Protocol in open systems specifications are decomposed into several defined parts as follows:

1. Protocol State Machine which enforces the transition rules.
2. Protocol States which define the legal actions of the protocol per state. Actions include parsing incoming PDUs. Constructing outgoing PDUs. Managing protocol state variables. Evaluating predicates. Note- The reception of a service primitive and the generation of dependent actions are considered to be an indivisible action as well as the reception of a PDU and the generation of dependent actions. The indivisible behavior can be/should be an implicit characteristic of protocol generated by the tool. This suggests the qualitative step granularity of the simulation.
3. Predicates which define state variable relationships. The state machine enforces the rules by evaluating the predicates.
4. Incoming events which are service primitives and timer events. Service primitives convey commands and messages, PDUs. Service primitives may have embedded PDUs, protocol data units (i.e. ASN.1 coded messages). Service primitives may also convey additional parameters for entities below or beyond the immediate neighbor. Timer events only have an abstract format in the standard. It is desirable to normalize timer events so they can be treated just like other input events. This suggests that timers be made manifest as input queue entries. The place in the queue defines its time relationship to message arrival.
5. Outgoing events. Events are usually organized with reference to the originator (protocol service user or provider)
6. Grammar - Structure and encoding of APDUs, application protocol data units. The grammar defines all legal combinations of message format. States define when each can be sent or received.
7. A queueing mechanism is implied by the fact that the FSM must be able to hold at least one message which can be parsed and another which can be constructed. Flow control of messages in the queue is also implied along with queue length. The tool should have a flow control building block even though the standard does not specify one.
8. FSMs communicate with each other through side effects of constructing messages and conveying the messages by way of a service primitive. Service primitives are abstract. Part of the implementation of service primitives can be queueing mechanisms. T his is implementation dependant. Note- the tool may define such implementation dependant characteristics as being fundamental to the tool design and to all protocol generated by the tool.

The above list is a good starting point upon which to formulate a set of protocol building blocks. It is not sufficient to implement a protocol. The tool will need additional building

blocks such as queue widgets, message parsing functions, field comparison operators, format handlers relative to a grammar, and ASN.1 encode/decode, etc.

3 ACPM State Transition Walk-through

The association control protocol machine has been select as an example to explore in detail protocol behavior. This exercise will help in discovering what is needed for protocol building block objects within APART. The example will cover the transition from idle (0) to Waiting AARE(1). The actions involved in going through these states are representative of what the tool needs to deal with in encoding, representing, and simulating state transitions. Either side of the communication could initiate the association. This example will cover the case in which the local user is the initiator:

1. The local user builds an A-ASSOCIATE request primitive and optionally, the Protocol Version and implementation information.
2. The local user somehow causes an instantiation of the Association control Protocol Machine, ACPM. In this protocol the instantiation could be bundled with the first primitive/message queuing, function. The tool somehow has to know which module/object to instantiate. Two mechanisms come to mind. Assuming the tool has a static description of FSM connectability it could determine which FSM to instantiate. A second approach would be for the user to explicitly identify which module and which instantiation. Such overt connection is necessary in some protocols which deal with session restart or multiplexing among several directly connected peers.
3. The ACPM tests the truth of predicated p1. p1 true is defined to be the state of the system in which the ACPM can support the requested connection. The tool needs the capability to associate a collection of logic, possibly arbitrary, to a predicate identifier. It may be necessary to provide a limited set of such predicates for tool user level use.
4. The ACPM checks the type of request primitive against the valid set which it should accept given the current state. If it is an invalid primitive some indication should be posted by the tool. Such action is not defined in the standard.
5. The ACPM constructs an AARQ APDU from values in the primitive. Twelve fields of information are defined in the AARQ. These fields need to be encoded in ASN.1 notation. This would have probably been done by the user already. It depends on the chosen format for the primitive which is not standardized. Some of the implied functions are the ability to allocated data space to hold the new APDU, the ability to manipulate ASN.1 field values (such as conversion from encoded to numeric or string form), the ability to build ASN.1 structures such as sets and sequences, the ability to check an incoming PDU for correct syntax relative to a grammar.
6. If the ACPM were receiving the AARQ APDU is would perform specific syntax checks. A given protocol is more or less liberal in such checks. The point is that

the tool must have the flexibility to modify the severity of checking. In particular ACPM defines rules of extensibility which state: undefined tagged fields or unknown bit names should be ignored while undefined APDU or fields constitute a protocol violation. For ACPM this means that extra undefined tagged fields or bit fields can be ignored, but required items must be present. In addition to syntax checks, field values may need to be compared to ACPM internal variables. The protocol version field is such an item. The ACPM must know which versions it will support. Upon receipt of an AARQ it must find/parse the version field. The field is encoded as a bit string. The string is decoded to some internal form which can be numerically compared against the acceptable values

7. It then encapsulates the APDU as the user data field in a P-CONNECT request primitive.
8. It packages for pass through additional parameters for presentation and session.
9. It sends the primitive down to the presentation service. In order to take care of the general case this has to be a queueing mechanism.
10. The ACPM changes it state from idle(0) to waiting-AARE (1).

The ACPM sets local state variables. Abstract state variables are sometimes called predicates. One ACSE predicate, p2, if true means that the ACPM originated the association. In other protocols such variables may represent expiration timers.